UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

Security analysis of hybrid Intel CPU/FPGA platforms using IOMMUs against I/O attacks

Sicherheitsanalyse von hybriden Intel-CPU/FPGA-Plattformen mit IOMMUs gegen E/A-Angriffe

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Christoph Jannik Peglow

ausgegeben und betreut von
Prof. Dr. Thomas Eisenbarth

Lübeck, den 30. Juli 2020

Abstract

This master thesis reviews the security of state-of-the-art hybrid platforms consisting of Intel Xeon CPUs and Field Programmable Gate Arrays (FPGAs). These are now increasingly found with cloud providers in „Infrastracture-as-a-Sevice“ models. Some Hardware interconnects, such as PCIe or Thunderbolt, offer the connected peripheral devices Direct Memory Access (DMA), which allows them to access arbitrary memory locations unmonitored. Input/Output Memory Management Units (IOMMUs) are a memory protection mechanism used to to prevent peripherals from abusing Direct Memory Access (DMA) to attack the CPU by adding an address translation layer. The IOMMU comes with additional translation caches to compensate for the negative performance impact of the added abstraction layer. We have evaluated a hybrid system with an active IOMMU in terms of its security against DMA attacks. Our investigations have shown that, in our threat model, a DMA attack is still possible, but its practical relevance has to be reviewed further. Furthermore, we tried to reverse engineer the I/O Translation Look aside Buffer (IOTLB), one of the translation caches, but were not able to guess the correct mapping function yet.

Diese Masterarbeit untersucht die Sicherheit modernster Hybridplattformen, die aus Intel Xeon CPUs und Field Programmable Gate Arrays (FPGAs) bestehen. Diese finden sich nun zunehmend bei Cloud-Anbietern in „Infrastracture-as-a-Sevice“ Modellen. Einige Verbindungsstandards, wie PCIe oder Thunderbolt, bieten den angeschlossenen Peripheriegeräten Direkten Speicherzugriff (DMA) an, der es diesen Geräten erlaubt unkontrolliert auf beliebige Speicheradressen zuzugreifen. Input/Output Memory Management Units (IOMMUs) sind ein Speicherschutzmechanismus, der verwendet wird, um zu verhindern, dass Peripheriegeräte DMA missbrauchen, um die CPU anzugreifen. Dies geschieht durch das Hinzufügen einer zusätzlichen Adressübersetzung. Die IOMMU verfügt über Übersetzungs-Caches, um die negativen Auswirkungen der Abstraktionsschicht auf die Leistung zu verringern. Wir haben ein Hybridsystem mit einer aktiven IOMMU im Hinblick auf seine Sicherheit gegen DMA-Angriffe bewertet. Unsere Untersuchungen haben gezeigt, dass ein DMA-Angriff in unserem Bedrohungsmodell immer noch möglich ist. Die praktische Relevanz des Angriffes muss allerdings noch weiter überprüft werden. Darüber hinaus haben wir versucht, den I/O Translation Look aside Buffer (IOTLB), einen der Übersetzungs-Caches, zu analysieren, waren aber noch nicht in der Lage, die korrekte Abbildungsfunktion zu erraten.

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

_____

Lübeck, 30. Juli 2020

Acknowledgements

First I would like to thank Prof. Dr. Thomas Eisenbarth from the Institute for IT-Security at the University of Lübeck, without whose supervision this work would not have been possible. I would also like to thank my personal supervisor, Thore Tiemann, for always providing great advice and helping me to the best of his ability, especially during the Corona pandemic. Last but not least, I would like to thank my proofreaders Leon, Malte, Sophie and Thore for their many helpful comments.

Contents

1 Introduction

Modern computers are a complex systems of interlocking hardware and software components. Especially important for many application scenarios are Input/Output (I/O) devices. These devices are used to process an input stream of data from the outside world and/or produce an output stream of data to communicate with it. I/O devices can integrated in the system internally or connected as external devices through various interfaces (peripherals). A few examples of classical internal I/O devices are network adapters, graphical processing units (GPUs) and field programmable arrays (FPGAs). Docking stations, projectors and even USB-C chargers are external devices.
To increase the performance of these devices, Direct Memory Access (DMA) allows them to fully access system memory. This enables the CPU to not take part in data transactions, between I/O devices and the main memory to keep working on other tasks. This, however, requires trust in the I/O devices because DMA transactions are not monitored. These devices even gain access to the operating system's internal data structures, placing the peripheral in the OS's Trusted Computing Base (TCB).
Since the early 2000s, multiple connection standards on the markets have arisen that make DMA available to external devices through FireWire, Thunderbolt 2 and now USB-C with Thunderbolt 3. This led to an increase of interest in DMA attacks like [Dor], [BDK05] and [Boi06], where the attackers were able to extract data from the victim's computer or executed privileged code through peripheral devices.
Having physical access to a system made these attacks easy to accomplish. And with the ongoing digitization of the economy, education, health care and many other areas, OS vendors and hardware producers acknowledged these threats and started developing countermeasures against such DMA attacks. One of the countermeasures is the *Input/Output Memory Management Unit* (IOMMU or I/O MMU). The goal of this countermeasure is to assign memory regions to peripheral devices and prohibit them from accessing anywhere else. This prevents malicious devices from snooping in memory and modifying other processes behavior. Something with a very similar functionality had already been developed in Memory Management Units (MMUs), which implement an abstraction layer for software applications. By translating the *physical addresses* (used by the hardware) of the RAM to *virtual addresses* (used by software processes), the software is only aware of its own reserved memory regions and can not access other applications memory.
IOMMUs are also used to assign I/O devices to virtual machines. Intel calls their imple-

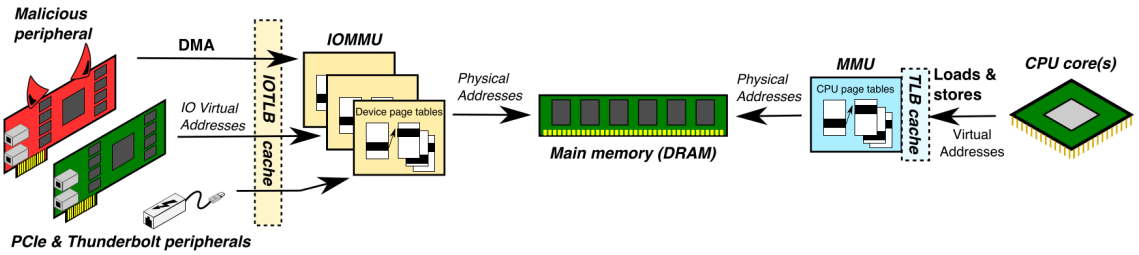

Figure 1.1: The IOMMU translates the I/O virtual addresses of memory requests to physical addresses and applies access control, similar to how the MMU translates virtual addresses from processes. (Taken from [MRG+19]).

mentation to support the virtualization of platforms *Intel Virtualization Technology* (Intel VT). A component of Intel VT is the IOMMU, called *Intel Virtualization Technology for directed I/O* (Intel VT-d), which is used to isolate and restrict device access to the resources assigned to that partition. This works in the same way as MMUs, just that the *virtual addresses* (VAs) here are called *I/O virtual addresses* (IOVAs). The translation process from IOVA to *physical address* (PA) is called *DMA remapping* and uses multiple levels of page tables. A full table lookup of an IOVA to a PA is slow, which is why an *I/O Translation Lookaside Buffer* (IOTLB) is used. This is a cache that stores recently translated address-pairs. Gras et al. [GRBG18] have shown that cache side-channel protections are defeatable by attacking the TLB of the MMU. As a result they were able to break multiple encryption schemes and extract plain text messages from a VPN connection. This idea is not only interesting because the authors were able to circumvent state-of-the-art memory protection mechanisms, but also because there was previously the assumption that the TLB was insufficiently reliable and too coarse-grained to work as an attack vector. We will investigate whether this assessment is also wrong for the IOTLB.

1.1 Goal of this Master Thesis

The goal of this thesis is to review the security of current hybrid platforms of Intel Xeon CPUs and FPGAs with enabled IOMMU. To validate this, we are going to verify the following theses:

- It is possible to implement a DMA attack, that leaks sensitive information or gives the attacker full control over the victim system, using the FPGA.
- The IOTLB is a viable covert channel for cooperating parties.
- The I/O Translation Look aside Buffer can be used as a timing based side-channel to attack other peripherals.

1.2 Threat Model

For our threat model we consider an Infrastructure-as-a-Service (IaaS) scenario that is common on rented cloud servers, which offer the tenant accelerators like FPGAs [Ama], [ECS]. These hybrid platforms of Intel Xeon CPUs and FPGAs have become increasingly popular over the last years. To be economically efficient and have high resource utilization, the service provider rents the same server to multiple customers so that their applications run in parallel, in their own virtual machines (VM). To isolate these VMs, a VM monitor (VMM), also called hypervisor, is used to virtualize and schedule system resources. Practice has shown, however, that this does not stop an attacker from extracting information, since the virtualized hardware is based on shared physical components like the RAM. This enables an attacker, that is capable to perform a DMA or cache side-channel attack on that machine, to spy on the other processes. Furthermore a DMA attack might also be able to manipulate program behavior by modifying the main memory. This violates several security goals such as confidentiality and integrity of data that a service provider should assure its customers.
The following criteria define our threat model:

1. It does not matter whether the FPGA is an external or internal device.

2. The FPGA has to be programmable by an authorized user and capable of sending memory read and write requests through a DMA-capable interface, e.g. PCIe, FireWire or Thunderbolt. Devices with only read access can also be used to attack a system, but this limits the gains of an attack to extracting data and does not give the ability to manipulate control flow or to gain full control over the system.

3. The user should only be able to execute unprivileged code on the connected CPU.

4. Users are not able to alter the OS, device driver, I/O controller or the functionality of the FPGA in such a way that it is not usable by other users or that it imitates a different device on the server as shown in [MRG+19]. In our case this is detectable behavior, because the expected device disconnects and another new device connects to the system if the peripheral is hot swappable. If it is not, this action would need a server restart, which should not be possible to an unprivileged user and alarms the service provider.

5. The attack should be transparent to the renter and other users.

1.3 State of Research

DMA attacks from external peripherals have been possible since FireWire ports were build into PCs and laptops. Some of the earlier attacks were presented in [Dor], [BDK05] and [Boi06]. Later, attacks over PCI, PCIe and Thunderbolt were also discovered [AD10], [BS12], [sr14]. Inspired by these attacks, generic open source DMA attack platforms like SLOTSCREAMER [FC] [Fit] and PCILeech [Fri] have been developed. These platforms can steal sensitive data, violate kernel security policies and even take full control of their victim through various hardware interfaces and operating systems. A big downside of these attacks and platforms is, however, that they require unrestricted access to system memory. This means that they do not work when faced with memory protection mechanisms, such as an IOMMU.

Over the last years, a few attacks with the capability of circumnavigating an IOMMU have been presented. However, many of them take advantage of the early boot phase of the computer where the IOMMU is not yet enabled or configured correctly. Many platforms enable DMA at boot time, before the OS is loaded. As a result the IOMMU is not setup when the attack strikes. Sang et al. [SELND10] proposed to manipulate the IOMMU page tables, ACPI tables and configuration registers, but did not show working exploits. This should have been possible however, as Wojtczuk et al. [WRT09] did show how to modify the ACPI tables during boot one year earlier, tricking the OS into believing there was no IOMMU present on the system. Morgan et al. [MEANK16], [MEANK18] then presented in 2018 a proof-of-concept, in which they exploited a vulnerability window during startup. In this window, they wrote a malicious context table into the RAM and then overwrote the OSs root table entry to point to their own malicious table. This resulted in the IOMMU handling every DMA request by their device in pass through mode, which means every access is granted.

In early 2019, Markettos et al. [MRG+19] presented Thunderclap, a novel hardware platform based on an FPGA, able to attack systems with IOMMU protection. Their approach was to attack the DMA interface between peripherals and device drivers, by abusing *spatial* and *temporal vulnerabilities* in the memory pages allocated by the driver for DMA access. Memory addresses are normally grouped into pages that can be reserved by a device driver for its connected device. However, drivers were able to allocate memory that was smaller than the smallest page size. Since the IOMMU mapping is page-granular, the peripheral always had access to the whole memory page. This resulted in a leakage of memory from other processes and even in kernel pointer exposure, which was abused to gain access to a root shell on the victim's system. Temporal vulnerabilities can appear when the device driver unmaps memory but the translation-data is still cached in the IOTLB. For as

longs as the cache entry is not invalidated, the device can still access that memory page, even when it is allocated to another process. The IOTLB invalidation process happens asynchronously on modern system to increase performance.
PCIe, a high-speed serial connection used in almost all modern computers, supports *Address Translation Services* (ATS). This feature comes with a severe security flaw by design and has been briefly mentioned by [MRG+19] and [Kup18]. The purpose of this service is for devices to be able to implement their own Translation Lookaside Buffer to bypass IOMMU translation and relieve stress from the IOTLB. A malicious device can simply set the necessary header bit for every memory request, implying that no address translation by the IOMMU is necessary. If this feature is enabled by the OS, an attacker can get unrestricted access to memory, even with enabled IOMMU.
Another focus of our work lies on cache attacks. They are the most prominent side-channel attack class and there are many different variants. Some examples are *Flush+Reload* [YF14], *Prime+Probe* [OST06] and *Evict+Time* [OST06]. Cache attacks take advantage of the fact that caches are shared between processes, even though the main memory is not, for isolation reasons. As a result, cache side-channel protections have been deployed to the shared CPU caches. These range from denying an attacker the ability to make precise timing measurements or partitioning the caches' sets [KPMR12] or ways [LGY+16].
Gras et al. [GRBG18] have shown how to defeat cache side-channels protections, such as CAT and TSX and were able to reconstruct EdDSA and RSA keys with high probability. They accomplished this by attacking the Translation Look aside Buffer (TLB) of the MMU, which was not protected by state-of-the art-cache side-channel protections and still shared by processes. To be able to carry out their attack, they reverse engineered the TLBs structure, i.e. the number of sets and ways, which are kept secret by Intel. They found out that, for modern generations of Intel processors, different mapping algorithms were used for the different levels of caches. After exploring how to monitor the cache line of the TLB, they trained a classifier that distinguishes the accessed TLB sets through timings.

1.4 Approach

We will start by implementing a simple DMA attack, that works with a disabled IOMMU, on our test platform to get an understanding of how to program the hardware and to see if other memory protection mechanisms are in place. In the second step we will enable the IOMMU and check whether our attack still works. We also take basic timing measurements and compare them to measurements with a disabled IOMMU. From that we will be able to derive the impact of the additional abstraction layer on the transaction time. This knowledge is needed when we start analyzing the IOTLB. Then we will analyze whether

the temporal and spatial vulnerabilities have been closed and if so, how. If the current state of the IOMMU is still vulnerable to these security flaws, we will implement attacks to abuse them and gain control over the victim's system. However, we will not use PCIe Address Translation Services to perform an attack.
After exploring the attack scenarios using, we want to focus on the IOTLB. We are going to try to replicate the experiments done by Gras et al. [GRBG18] in the context of IOTLBs. Since the functionality of TLBs and IOTLBs is basically the same, we hope that a very similar mapping function from IOVA to cache set is used. If we are successful in reverse engineering our IOTLB and creating multiple eviction sets, we want to build a covert channel through which cooperating parties on the same machine can communicate. Lastly we want to show that the IOTLB can be used as a coarse-grained side channel to attack other peripherals.

2 Background

This chapter provides the necessary background needed to understand the approach in the following chapters. First, we describe the Intel computer architecture, focusing on the important components needed for DMA. Then we explain the basics of PCI and PCIe, followed by an overview over DMA and how it has been abused in the past to attack systems. After that we introduce Memory Protection Mechanisms and explain how Intel's countermeasure against such DMA attacks, the *Virtualization Technology for Directed Input/Output*, works. In the last part of that section we introduce translation caches. To conclude this chapter we present the hardware we used in our experiments.

2.1 Computer Architecture Background

To understand the communication between CPUs and peripherals, it is important to grasp the basics of modern computer architecture. The information in this section is taken from *Intel SGX Explained* [CD16], unless specified otherwise.
The backbone of a modern computer is the *motherboard*. It connects all the resources of a computer, beginning with the *memory* (RAM) and *processors*, as well as all kinds of Input/Output devices (I/O devices or peripherals) like keyboard and mouse, display adapters for monitors and network interface cards (NIC). These peripherals are used by the CPU to communicate with the outside world. Internal communication takes place via different bus systems. Most buses on the motherboard come together in a Platform Controller Hub (PCH), also known as chipset or I/O Controller Hub. This companion chip is an evolution of the former *southbridge*, while the former *northbridge* was mostly integrated into the *uncore* region of the CPU, as seen in fig. 2.1 [Tur14].
The PCH houses the controllers for many interfaces like *Universal Serial Bus* (USB), *Serial ATA* (SATA) and *Peripheral Component Interconnect* (PCI). For bus systems that are DMA capable, part of their controller can be a Direct Memory Access Controller (DMAC) (see section 2.4). The Intel 200 and Intel Z370 Series Chipset Families Datasheet, for example, list a total of 4 DMA controllers for GSPI, I^2C, UART and the Integrated Sensor Hub (ISP) [Int17] [Int18]. The PCH is connected directly to the CPU via a point-to-point serial link, known as *Direct Media Interface* (DMI). DMI is used as a high-speed bus to allow quick data exchange between the CPU and the I/O devices connected to the PCH. PCI and *PCI Express* (PCIe) will be introduced in section 2.2.

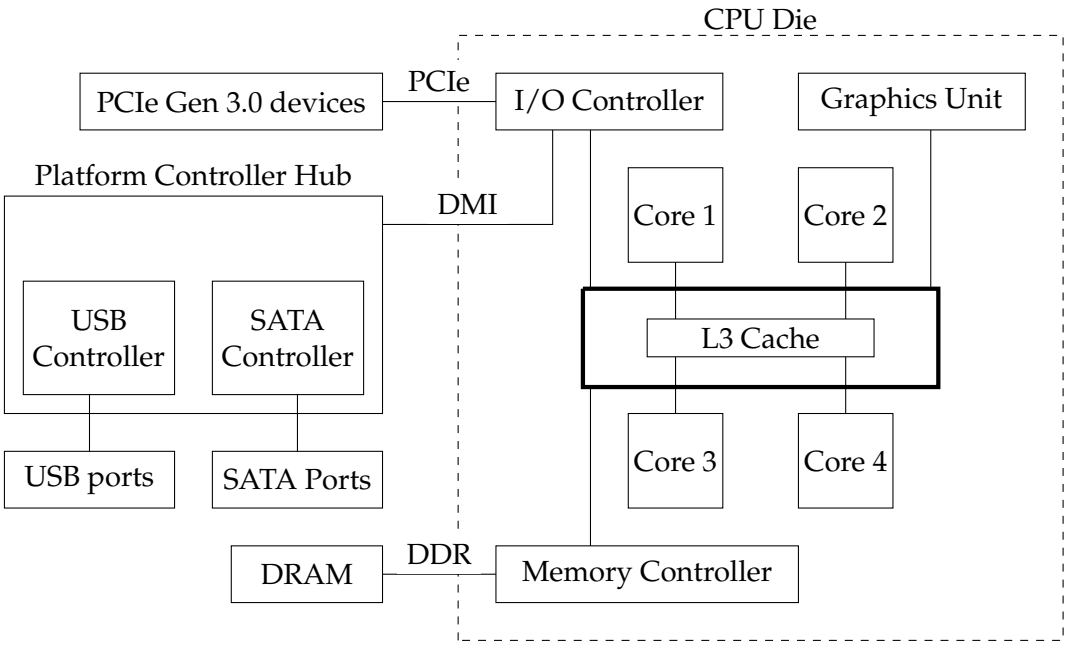

Figure 2.1: This figure is a simplified representation of the uncore region of an example Intel CPU die and shows the connections between components. (Inspired by [CD16, 17] and [Tur14])

.

2.1.1 I/O Address Spaces

This subsection describes the different I/O address spaces present in the Intel PC architecture and how the CPU and I/O controllers interact with it. These are necessary for sharing information between the processor and a peripheral since the information only needs to be stored in the shared address space, as well as the configuration of I/O controllers.

Programmed I/O space

In the past, the CPU had to communicate with I/O controller registers through special I/O instructions. These allow the processor to read and write to registers mapped into the *Programmed I/O space* (PIO) [SND11]. With this, the CPU was able to configure peripheral devices through that shared address space. Some devices still use this for backwards compatibility.

Memory Space

The *memory space* is an address space, where either main memory or I/O locations can be mapped to. It is also called *memory-mapped I/O* (MMIO). These I/O locations can be configuration registers of controllers or peripheral devices, as well as internal memory of these peripherals. The CPU can access this address space like regular memory by sending a memory request to the I/O controller. This request is then forwarded either as PCIe or DMI to the respective controller. Furthermore, peripheral devices can also access the mapped memory locations through DMA (see section 2.4). This is used by many I/O devices, like ethernet adapters, graphics cards and FPGAs.
The configuration space, specific for PCI, PCI-X and PCI Express devices in the Intel PC architecture, will be explained in the next section.

2.2 Peripheral Component Interconnect

Peripheral Component Interconnect is a bus mastering system. This means that the bus is a broadcast network connecting multiple peripherals with an I/O controller. Since there can only be one sender at a time on the network, a bus master has to be assigned. Only the bus master is able to communicate over the bus and if another device wants to take the role of master, the right has to be requested and only after receiving permission, the device becomes the new master.
Every PCI compliant device has to have a configuration space that is accessible by system configuration software at all times. This address space is divided into a predefined header region and a device dependent one. The header consists of fields that uniquely identify the device and allow the OS to generically control and configure the behavior of the device. This is used, for example, to detect every connected PCI device at system startup through a vendorID, a deviceID and a revisionID.

2.3 Peripheral Component Interconnect Express

Peripheral Component Interconnect Express on the other hand is a packet-based high-speed serial point-to-point connection between every PCIe slot and the PCIe root complex. It consists of several independent links or lanes between endpoints(x1, x2, x4, x8 and x16), with every lane offering 8 GT/s (Giga Transactions per second) in the version 3.0 [PCI10]. Even though the architectures of PCI and PCIe are quite different, some of the core attributes of PCI have been maintained and result in backwards compatibility. An exemplary topology of PCIe is shown in fig. 2.2.

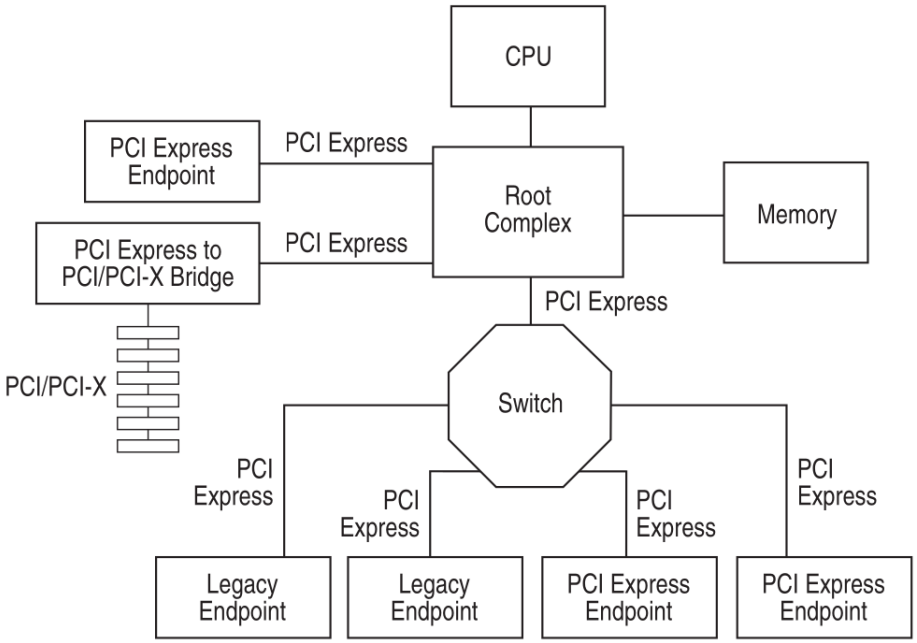

Figure 2.2: This shows an exemplary PCIe topology (taken from [PCI10, 41]).

There are three layers in total: physical, data link and transaction. The data link layer is used for error correction, flow control and acknowledgments, while the transaction layer turns user application data into transaction layer packets. Each of the layers adds its own header to the packets, which results in varying overheads depending on the data size [NAZ+18].
Over the last couple of years, the Root Complex (RC) was moved from the PCH to the uncore region of the CPU die. This enables closer interaction between peripheral devices connected through PCIe and the CPU caches. This resulted in the development of Direct Cache Access (DCA), also known as Data-Direct I/O (DDIO) [KGA+20]. However, the host controllers of other expansion slots like SATA, USB or M2 are still connected by PCIe [MRG+19].

2.3.1 Mechanisms that impact PCIe Performance

In a few cases the physical address space seen by the CPU and an I/O device is not equivalent, because the Root Complex can add an abstraction layer. The targeted address of a memory request is first processed by the RC, resulting in the correct physical address. Address translation can significantly increase the overhead of a memory request through multiple memory accesses (page table walk). Translation caches are used to mitigate the impact. These caches are very small, however, and need to hold translations for multiple

peripherals. To alleviate some of the pressure off of the IOTLB, *Address Translation Services* (ATS) [PCI09] were developed. These give ATS capable devices the opportunity to implement their own translation look-aside buffer. Requests from such a device need to set a specific bit in the PCIe transportation layer header of memory read and memory write packets and are then passed through the root complex and not processed. This results in lower overhead and better DMA latency.
The IOMMU (see section 2.6.1), a modern protection mechanism against I/O attacks that translates I/O virtual addresses to physical addresses, is located between the host and the root complex. It can impact the latency of PCIe transactions quite severely, as was shown in [NAZ+18]. The IOTLB is part of the IOMMU and caches recently translated addresses in order to compensate for the negative impact. Nevertheless, the latency now depends not only on the state of the CPU caches, but also on the state of the IOTLB (see section 2.6.2). However, requests with the ATS bit set are passed through the IOMMU without going through the translation process [MRG+19]. Communication between peripherals, called peer-to-peer transactions, are also not affected by address translation, because the communication does not pass through the IOMMU. For this kind of communication, the PCI MMIO address space is used instead of IOVAs [lin].

2.4 Direct Memory Access

In many applications the acquisition of data from peripheral devices has to be fast. The three main transfer mechanisms for data acquisition are polling, interrupts and DMA [Har94]. The first two mechanisms revolve around transferring the data between an I/O device and the memory using the CPU. When polling data, the CPU repeatedly checks for new data to arrive from the peripheral device. This prevents it from running other parts of the main program and keeps the CPU busy. If interrupts are used, the CPU has to configure an interrupt handler, which notifies the CPU when an I/O device has new data. With this, the processor can work on other tasks and can interrupt work whenever an interrupt is raised to move the data to a buffer for later processing. The polling data transfer mechanism can occupy the CPU for especially long times, but there are also scenarios where interrupts do the same. This results in a slowdown for all processes running on that machine. A basic example for CPU interaction is shown in fig. 2.3.
Direct Memory Access (DMA) was developed to reduce the amount of CPU cycles needed to transfer data and to increase the data transfer rate. As the name suggests, peripheral devices can get direct access to the systems main memory by taking control of the system bus [CS15].

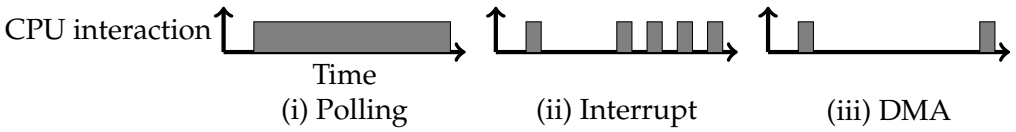

Figure 2.3: Shows how much CPU interaction is needed to move data between memory locations depending on the used method.

There are two different variants:

1. Bus mastering (first party DMA)

2. Third-Party DMA

In a first-party DMA system, a peripheral has to bring its own DMA engine. After the I/O device configures its engine, that engine requests the bus master role and initiates the DMA transfer with the memory controller over the system bus [YZZ17]. When the transfer is completed, the device can send an interrupt to the CPU to notify it about the finished transfer. This notification is not necessary since the transfer is already done and the peripheral can just give the bus master role back to the CPU. The transaction then happened transparently from the processors point of view [Har94]. This is not compliant behavior, but was used by DMA malware to secretly extract data [SB13].

Third-party DMA is realized by a dedicated piece of hardware called *Direct Memory Access Controller* (DMAC). This device can be programmed by the CPU (slave mode) or an I/O device (master mode) to perform a sequence of data transfers. These can occur directly between memory locations [AAA19] or from a peripheral device to memory and vice versa [Har94], [Anu16]. The DMAC is often part of the underlying bus systems controller, which means it is closely connected to the CPU, as seen in fig. 2.1.
DMA results in a very low latency between the data acquisition device and the memory, because the I/O device can initiate the data transfer as soon as the data chunk is acquired. This also minimizes the storage needed on peripheral devices. One of the biggest advantages of DMA is that there is almost no expenditure of CPU cycles, since the CPU only has to configure the DMA controller to start and stop transmissions. Conventional data transfer requires the CPU for a large part of the transfer, blocking the CPU from working on other tasks. This becomes especially interesting for small computers and microprocessors, because of their low clock frequency [CS15]. Examples of common hardware interfaces that support DMA are Peripheral Component Interconnect (PCI/PCIe), FireWire (or Firewire, i.LINK) and Thunderbolt [MRG+19].

2.4.1 Sequence of a third-party DMA transaction

Any device on the system bus can issue a DMA transaction [AD10]. The DMAC often has multiple DMA channels for the different device slots on that bus system [Har94]. The number of channels depends on the DMAC. Over these channels, a peripheral can trigger a DMA transfer by sending a DMA request (DREQ). These work like interrupts and the DMAC can prioritize the requests from different channels and generates the address and control signals for the system bus [Int15].
If the CPU wants to transfer data from RAM to a peripheral device, it writes the starting address and the length of the data block, as well as the transfer mode into the internal registers of the DMAC. Then the mask bit of the corresponding DMA channel will be cleared. This will notify the controller about the DREQ by the CPU. After receiving the request, a Hold request (HRQ) will be send to the CPU to gain hold of the system bus. The Processor grants the system bus with a Hold acknowledge (HLDA) signal. Now the DMA transfer begins as the DMAC outputs the data address, a DMA acknowledgment on the DMA channel of the requesting device and simultaneously pulsing memory-read (MEMR) and I/O-write (IOW) signals [Int15]. Transfer will be initiated by the device controller and the memory controller [Fal14]. The data will then be send over the system bus without passing through the CPU or the DMAC.
A data transfer from an I/O device to memory works almost the same as the previously mentioned process. Here, the signal is raised by the I/O device and the DMAC takes the master role on the system bus. Again, a HRQ and HLDA have to be exchanged and then the data addresses and control signals will be generated by the DMAC. However, instead of MEMR and IOW, memory-write (MEMW) and I/O-read (IOR) will be sent [Int15].

2.5 DMA Attacks

With the increasing popularity of FireWire in the early 2000s, DMA attacks from external devices became possible, because FireWire ports were present on the outside of the PC case, which makes them easily reachable. Nowadays, Thunderbolt and USB-C are the prevalent DMA capable connectors on the outside of the case. This fact in addition to the unsupervised, full access to the host system memory gives an attacker the ability to plug in a malicious DMA-enabled device into a victim's computer and initiate a DMA transfer from the main memory. The CPU ceases control of the system bus to the peripheral and does not check if the requested memory addresses belong to memory regions that are relevant to that device, or if they contain critical system code or data structures. This allows an attacker to read and write arbitrary data from memory, bypassing security mechanisms of the operating system and lock screens [For11], [AD10]. Thus, an attacker is given almost

countless possibilities to compromise its victim, such as stealing data and cryptographic keys, installing and running spyware, as well as modifying the (operating) system and internal data to allow backdoors and other malware [SB13], [YZZ17].
Sang et al. [SND11] propose the classification of DMA attacks into (1) attacks that access main memory and (2) peer-to-peer DMA attacks that access other peripheral's internal memory regions through memory mapped I/O (MMIO).
From the former category, an early DMA attack was presented by M. Dornseif at the PacSec 2004 in Japan [Dor]. He and his colleagues modified an iPod with the Linux distribution Knoppix and were able to extract data from the connected Mac's memory. In another demonstration, Dornseif showed how to alter the content of the monitor by manipulating the framebuffer of the graphic controller via a FireWire connection [SND11]. This attack is categorized as a peer-to-peer DMA attack, since the DMA targets another peripheral and its memory.
There are open-source tools like PCILeech [Fri] that can successfully attack systems using DMA attacks through external USB3380 or FPGA hardware. PCILeech is capable of accessing the RAM and the file system, as well as removing the login password requirement, executing code and spawning system shells. Currently supported target systems are UEFI, Linux, FreeBSD, MacOS and Windows.
A recent attack was presented by Markettos et al. in early 2019 [MRG+19]. They developed an open source hardware platform, called Thunderclap, which is suitable for use with internal PCIe slots and external Thunderbolt 2 and USB-C ports with Thunderbolt 3. All that was needed for their developed platform, disguised as a trustworthy device, was to be connected to the victim's PC. Doing this, they managed to circumvent the state-of-the-art protection mechanism described in section 2.6.1. The authors found that the IOMMU is disabled in almost all of the tested operating systems except MacOS. These are Windows 7.1, 8.1, 10 Home/Pro/Enterprise, Ubuntu 16.04, Fedora 25, Red Hat Enterprise Linux 7.1, FreeBSD 11 and TrueOS 10.3. Secondly, by imitating a fully functional peripheral, all the systems were compromisable to *spatial* (see fig. 2.4) or *temporal vulnerabilities*, even with IOMMU protection. All the found security vulnerabilities were severe and some of them even allowed the authors to fully control their victim.

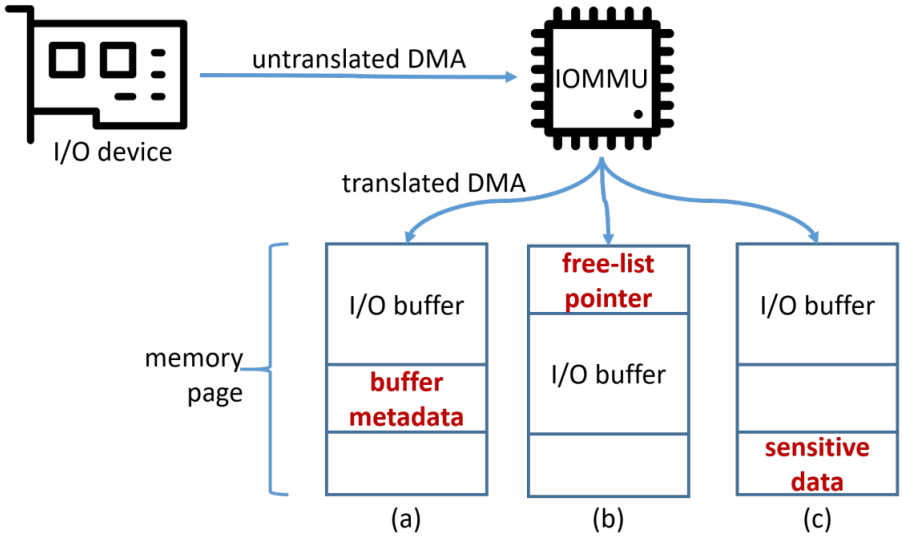

Figure 2.4: Spatial vulnerabilities occur when the I/O buffer is smaller than the page size and there lies other data in the same memory page such as (a) I/O buffer metadata, (b) free-list pointers or (c) other sensitive data (taken from [Kup18, 16])

2.6 Memory Protection Mechanisms

There are multiple ways for a platform to prevent DMA attacks. One example is the PCI Bus Master Enable (BME) bit included in the PCI and PCIe specifications [PCI10, 589]. This bit controls whether a PCI Express Endpoint can issue a DMA transaction, but comes with several limitations. First of all, the issuing device has to be a PCI or PCIe device, since this is a feature of that specific bus system. Furthermore, the BME needs to be respected by the peripheral device for it to work. But since the device may be malicious, it can simply ignore that bit and continue the DMA transactions. The BME for a PCI root bridge can be disabled to prevent this from happening, causing all connected devices to that bridge to not be able to issue further memory and I/O read/write requests [YZZ17]. This might be useful during the pre-boot phase, because the settings for BME are managed by the BIOS/UEFI and no operating system interaction is required. However, that is not a granular approach.

The access of memory by applications in user-mode is managed by an *Memory Management Unit (MMU)* in modern operating systems. This prevents an application from accessing physical memory addresses directly and thus prevents processes from accessing data that belongs to another application. This is realized by giving programs a range of logical addresses, which are mapped to the physical addresses and are not known by the application process.

Manufacturers adapted this idea as a countermeasure against DMA attacks, that are called *Input-Output Memory Management Units* (IOMMU). Intel's implementation of an IOMMU is called *Virtualization Technology for directed Input/Output* or Intel VT-d for short. AMD has their own *AMD I/O Virtualization Technology* [Adv16] and ARM the *ARM System Memory Management Unit (ARM SMMU)* [ARM16]. In this work we will only investigate Intel VT-d.

2.6.1 Intel Virtualization Technology for directed Input/Output

The information in this section, including section 2.6.2, are all taken from the architecture specification of the Intel VT-d [Int19b], unless specified otherwise.

The Intel Virtualization Technology for Directed Input/Output is part of the *Intel Virtualization Technology*, which consists of components that support running multiple operating systems and applications in independent virtualized partitions. To do this, it is necessary to isolate and restrict the access of these VMs to the resources of the partition that manages the device. It was later adapted to also prevent I/O devices from being able to access the whole system memory, since a countermeasure for DMA attacks was needed and the implementation of isolating VMs was also applicable to peripheral device isolation.
Intel VT-d provides I/O virtualization software with the following capabilities:

- *I/O device assignment*: enables the host system to adaptively assign I/O devices to Virtual Machines and making execution of I/O operations possible.
- *DMA remapping*: is used to translate physical addresses to logical addresses for Direct Memory Access from devices and vice versa.
- *Interrupt posting*: to support direct forwarding of virtual interrupts from devices and external interrupt controllers to virtual processors.
- *Reliability*: for recording and reporting of DMA and interrupt errors to system software that may otherwise corrupt memory or impact VM isolation.

The following sections will concentrate on DMA remapping and the translation process.

2.6.2 DMA remapping

Domains are isolated environments in the platform, to which a subset of the physical memory of the host is allocated. Every I/O device that has DMA will be assigned to at least one domain, but can be flexibly switched to others by the OS. Isolation of a domain is ensured by the *DMA remapping hardware unit* (DRHU), which is the part of the IOMMU responsible for the address translation. It receives every DMA request to memory and uses page tables to verify if the requested address is part of the domain of the requesting device. If it is not, the DRHU blocks the request entirely. DMA remapping also allows the host OS to allocate memory to I/O-devices that is not contiguous in the physical memory, but appears cohesive to the devices [YZZ17]. The Intel VT-d architecture specification allows DMA remapping to be done by software or hardware, but we have not seen or read about a single software implementation while researching this topic. A hardware module is simply faster than a software solution and access times to memory determine how fast an I/O device can operate.
Besides *Domain Isolation*, DMA remapping is also used by the operating systems in other ways, for example for *OS Protection* by implementing a *DMA Protected Range* (DPR) and *Protected Memory Regions* (PMRs) [Int20]. The former is a region of contiguous physical memory, ending right before the SMRAM segment and protected from all Direct Memory Accesses. However, this is only used to protect specific data structures, since it is a fixed memory range. The PMRs are two memory ranges of physical addresses that are also protected from DMA and can be freely placed in memory. They are frequently used to protect the page tables storing the address translation data. Nonetheless, PMRs can be used without address translation.
These protection mechanisms allow the operating system to block DMA to memory containing its critical code and data structures. This makes the OS more robust to incorrect or malicious programming of devices. It also enables *Feature Support* for legacy devices, which can only address 32-bit of memory and need to access memory above 4GB. For this, DMA remapping allows the translation of 32-bit memory addresses to 64-bit, since the I/O page-tables can remap the DMA request to high memory. And lastly, it is also possible to share the virtual address space of application processes and I/O devices in a *Shared Virtual Memory* scenario. This allows programs to use devices, such as graphics processors or accelerators, to speed up data processing without overhead.

DMA Request

The DMA remapping hardware unit splits incoming DMA requests into two categories: Requests with and without *process-address-space-identifier* (PASID). The Requests-without-PASID only submit the necessary information, such as type of access (read or write), targeted DMA address and size, and a *source-id* identifying the originating device of the transaction. This is implementation specific, since some I/O bus protocols have such identifiers build into their protocol while others do not. PCI Express devices have this feature integrated in the PCIe transaction layer header. The source-id is a 16 Byte value, specifying the exact bus [15:8], device [7:3] and function [2:0] of the originating device. Requests-with-PASID supply the same information, but on top also specify the targeted process address space identifier and optional attributes such as Execute-Requested flag and Privileged-Mode-Requested flag.

Address Translation

Understanding the address translation is crucial in finding attack vectors against the IOMMU, as it is the main proctective component against DMA attacks.
The translation process of a DMA Request is split into two parts. First the DRHU has to decode the domain the requesting peripheral belongs to and get its translation structure. Then the requested address has to be translated to a physical address using the found structure. For that a radix tree structure of different tables is used. Each of these table entries contains the address of the next table, or the physical address in the last layer.
Since there are many possible combinations of DMA requests, the requested pagesize and the amount of levels of page-tables used differs. We explain the scenario in this subsection that is applicable to our later experiments. The interested reader is referred to the Intel VT-d architecture specification for all the scenarios [Int19b, 30ff.]. First, we explain how a PCIe device is mapped to its domain in legacy mode (fig. 2.5) and then show how the requested memory address is mapped to a 4 KiB memory page (fig. 2.6).

To identify the domain of the requesting device, the *source-id* of the PCIe transaction is used. Starting point for this identification is the so called Root-Table, whose address is stored in the Root Table Address Register (RTADDR REG). The Root Table contains 256 entries for the number of possible PCI buses. It is indexed by the bus number of the *source-id* field in the request. The identified entry contains the address to the Context-Table of that specific bus. The Context-Table also contains 256 entries and is indexed by the device and the function number of the *source-id*. The corresponding entry points towards the second level page-table, which is the address translation structure for that

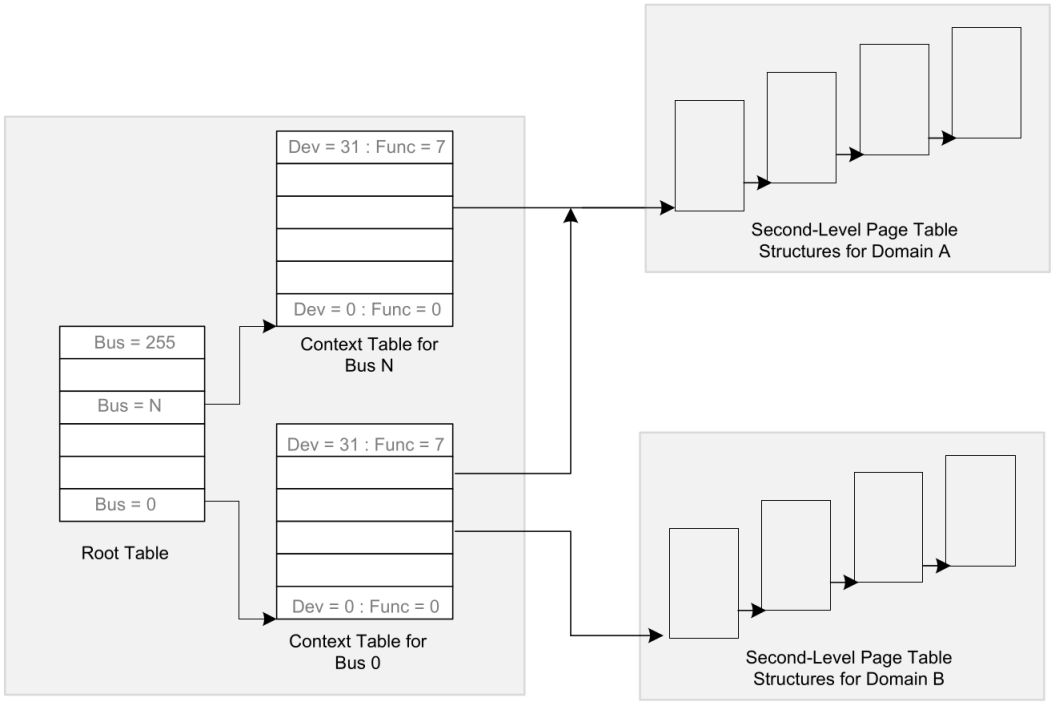

Figure 2.5: This shows how a requesting device is mapped to its domain. (Taken from [Int19b, 30])

specific domain. Multiple devices can be assigned to the same domain by simply pointing to the same second-level page table from the context table. For Requests-with-PASID there are two more levels of tables added after the context table.

The second level translates the 64-bit DMA address into a physical address pointing towards an entry inside a 4KB, 2MB or 1GB big memory page, also called page frame. The address is taken from the request and split into blocks, as seen in fig. 2.6. In our scenario, we have a 4-level paging structure in which the bits 63:48 are not used. The next 9 bits [47:39] are shifted by three to the left and added to the paging structure pointer from the first translation step. This entry contains a physical address pointer to the next table. The next 9 bits [38:30] are then shifted by three to the left and then added to the pointer. The resulting address points to the next table's entry and so on. With every table, the mapped memory region gets smaller until a table entry with the page-size bit set to 1 is found. This means that the physical address in that entry points to the beginning of the actual page frame the peripheral wanted to access. To now find the specific requested page frame in that page, the frame offset, i.e. the N unused bits [$N-1$:0], are added to the page address from the last translation table.

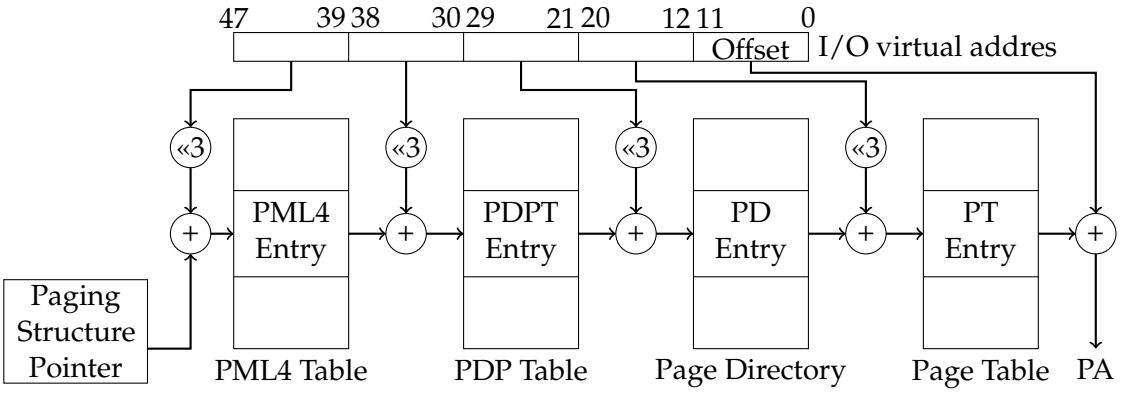

Figure 2.6: This shows the Address translation structure for a 4 KiB memory page.

Access Rights

The access rights for a memory page, meaning if it can, for example, only be read or also written, are also stored in these translation structures.
Translation faults can occur on two different reasons. Either the requested address has no valid translation in this domain or there is a valid translation but the access rights for that address prevent the access. To check these is also the job of the DMA remapping hardware unit. The access rights to a page depend on the attributes of the request and the rights specified by the paging-structure entries, that are accessed during the translation. Protection flags in the paging structure always outweigh the fields set in the request. For example a write request to a Write-Protect-Enabled (WPE) memory page will always be prohibited, if not every paging-entry in the translation structure grants the exception for this specific device.
Detailing all possible combinations of access types goes beyond the scope of this thesis. A detailed explanation can be found in the Intel VT-d architecture specification [Int19b].

Caching of Translation Data

Repeatedly conducting the address translation (*page table walk*) results in a huge overhead, because there are multiple memory accesses needed to find the necessary page tables, which in turn increase the latency of DMA transactions. To speed up the translation process, DRHUs cache various translation data in a Context-Cache, PASID-cache, Paging-structure Cache or *I/O Translation Look-aside Buffer* (IOTLB), depending on the exact hardware support and configuration. The most interesting caching structure for us is the IOTLB, as we want to use it to implement a covert channel and launch a timing based side-channel attack through it. The IOTLB stores the mapping from a I/O virtual ad-

dress page number to a physical frame, together with additional information about access rights.
It is important to note that the hardware does not ensure consistency between the cache structures and the actual translation structures in main memory. This means that the operation system has to invalidate cache entries on its own. It is fairly slow to invalidate a cache entry every time the translation structure changes (so called *strict mode*). From a security standpoint this would be the best approach, however, it is not done for performance reasons. Instead, the current Linux distributions have *deferred mode* activated by default, meaning that TLBs are getting invalidated at specific events or after a timeout occurs. This results in a timing window where invalid cache entries remain in the cache until the next invalidation cycles starts.
The Intel VT-d Specification [Int19b] mentions additional functionality that can be implemented for IOTLBs, such as also storing the source-id of the request that allocates an entry. However, these are only optional functionalities that hardware manufacturers can implement. The exact architecture of the IOTLB is thus dependent on the producer and mostly kept secret. In the next section we explain the basic principles of caches and then present the results of Gras et al. [GRBG18] in reverse engineering the TLB of the Memory Protection Unit.

2.7 Caches

The CPU uses multiple levels of cache to keep copies of previously fetched items from memory close to the cores in Static RAM (SRAM). Modern cache architectures use three Levels to compromise between speed and size of the cache levels. The Level 1 cache is split into an L1 instruction cache and an L1 data cache and the smallest of the three levels, but has the fastest access time, because it is closest to the CPU. Thus, it is the highest level cache. The Level 2 cache is shared for instruction and data fetches and larger than the L1 cache, but also slower. Every core has its own L1 and L2 cache. The L3 cache, also called Last-Level-Cache (LLC), however, is shared between all the cores. It is the largest and slowest, but plays an important role in cross-core access. Caches can be inclusive, which means that the higher level cache is always a subset of the next lower level cache.
Whenever data from memory is requested, the CPU first checks, if the data is present in L1 cache. If this results in a *cache miss* the L2 and if necessary the L3 cache is checked. Only if this third check also results in a cache miss will the data be fetched from main memory. If the data was not cached, it will be stored in one of the cache levels to speed up reoccurring requests.

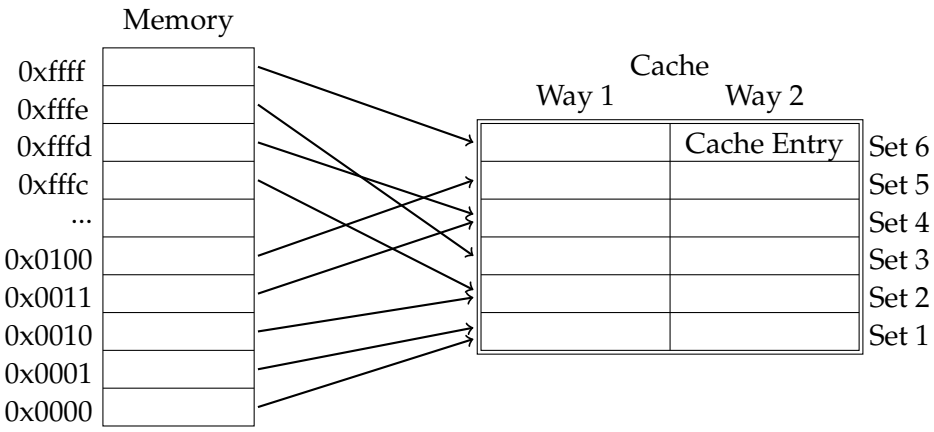

Figure 2.7: Shows an exemplary mapping between memory and a 2-way set-associative cache.

A cache consists of *sets* and *ways*, as seen in fig. 2.7. Each block of memory is mapped to a specific cache set, derived by a mapping function hashing the physical address of the block. When a cache set is full and a new entry is fetched another entry has to be evicted. This is done according to the *replacement policy*. Common strategies were Least-Recently-Used (LRU) or First-In-First-Out (FIFO), but nowadays more complex policies are usually used. The way, in which the address is stored, is the next free way or where the eviction policy clears the next entry. Because the caches are much smaller than memory, multiple lines of memory are mapped to the same cache set. There are three different kinds of caches. A *directly mapped* cache has only 1 way. This results in every memory address having one specific cache set where it can be stored. This makes these caches very easy to implement, however, it reduces the efficiency of the cache as sets might be unused, while others have to be swapped regularly. The opposite of a directly mapped cache is a *fully associative cache*. This means that the cache has only 1 set but many ways. This ensures maximum efficiency, as all cache entries are used. The disadvantage of this is the high effort needed to find the correct entry, since many entries have to be checked. The compromise between those two kinds is the *set-associative* cache. This is the most widely distributed kind of caches. A cache with 4 ways and multiple sets is called *4-way set-associative cache*.

Caches are shared micro-architectural components between processes. They increase the efficiency of those processes, but being shared violates process isolation on the micro-architectural level [KGA+20]. Attackers exploit caches for timing based side-channel attacks. An example is the Prime+Probe attack [VKM19]. Through creating *eviction sets*, i.e. collecting a set of physical addresses, of size of at least the number of ways, that all map to the same cache set, an attacker is able to bring cache sets into a controlled state. After

the victim's process is executed, the attacker accesses their eviction set again and are able to measure the timing differences and thus learn which cache sets were accessed by the victim's process.
There have been multiple ideas and implementations on how to protect caches against side-channel attacks. One proposed way is partitioning the caches' sets [KPMR12]. This can be done entirely in software, because the mapping of a memory block to cache set depends on the physical address of the block. The memory allocation subsystem of the operating system can be used to allocate memory to processes in such a way that concurrently running processes do not share cache sets. Another proposition is partitioning the caches' ways [LGY+16]. Most of the time there are less ways than sets, this is mostly realized with two domains, one for processes with the need to be run securely and one insecure domain. For this hardware support is needed, like Intel CAT.

2.7.1 Translation Look aside Buffer

Here we present the knowledge about the Translation Look aside Buffer. Since the IOTLB performs the same functionality, but in a different context, we base some of our assumptions in the later parts on this knowledge.
For TLBs of the MMU there are only two levels. The L1 cache, just like in the CPU caches, consists of a part for caching translations of code pages (L1 iTLB) and a part for data pages (L1 dTLB). The L2 cache is shared for data and code translations (L2 sTLB) and bigger than the L1 cache [GRBG18].
Common page sizes of memory are 4 KiB (i.e. $2^p = 4096 \rightarrow p = 12$) and 2 MiB ($p = 21$) for *huge pages*. The p-least significant bits of the physical and virtual addresses are the same. This due to the fact that the physical frame, resulting from a very similar translation structure as shown in fig. 2.6, is page aligned, i.e. its last p bits are zeroes. The page offset is added to the *physical frame number* to point towards the accessed memory address inside that frame [VKM19]. Larger memory pages use the TLB more efficiently, because they require fewer entries for mapping the same amount of memory [LYG+15].

2.8 Our Hardware

We used the existing hardware in the Institute for IT-Security as it meets the requirements of our threat model. Our test system uses two Intel Xeon Silver 4114 Server CPUs of the Skylake architecture, with 10 cores and 20 threads each. We have 640 KiB of L1, 10 MiB of L2 and 13 MiB of L3 cache, as well as 96 GB of RAM. This is installed on a Dell PowerEdge R740 motherboard. We are running the linux kernel version *4.4.0-185-generic* on the Ubuntu Server 16.04.6 LTS distribution. This is due to the fact that we have version 2.1 of the *Intel Acceleration Stack for Intel Xeon CPU* installed, which does not run on newer versions of Ubuntu. To synthesize our hardware designs we use Quartus Prime version 17.1.
The FPGA we are using as our malicious peripheral is an Intel Arria 10 GX 10AX115N2F40E2LG FPGA in the form of a PCIe expansion card clocked at 200 MHz.

3 Implementing a DMA Attack

In this chapter we present our implementation of a DMA attack and the insights we gained from taking multiple timing measurements.

3.1 DMA Attack with IOMMU disabled

First of all we wanted to check the statement of Markettos et al. [MRG+19] that the IOMMU is disabled by default on current Ubuntu Server distributions. We checked this on the Versions 16.04 LTS, 18.04 LTS and 20.04 LTS and found that the IOMMU is indeed disabled by default. This makes every system with these versions installed a potential victim to the most basic DMA attacks.
To check if other countermeasures against DMA attacks are available we kept the IOMMU disabled and implemented a basic attack on our Intel Arria 10 FPGA using the Core Cache Interface. (CCI-P) [Int19a]. CCI-P is a host interface bus intended for connecting an Accelerator Functional Unit (AFU) to an FPGA Interface Unit (FIU). The FIU is a platform interface layer, acting as a bridge between interfaces like PCIe, UPI and AFU-side interfaces such as CCI-P.
On the processor side, we used the OPAE C library [Cor] in our software to communicate with our designed AFU. The library provides abstraction for Intel FPGA resources by removing hardware and OS specific details and making them accessible from software programs running on the host.

3.1.1 Hardware Implementation

We started with the Hello World tutorial sample from the Intel FPGA Basic Building Blocks GitHub repository[GBAS], since it has almost everything that is needed to read and write memory locations from the peripheral. The addresses have to be provided by the software implementation presented in section 3.1.2. We expanded the functionality of the hardware design by adding an MMIO register to control whether the AFU reads or writes and also added a clock counter to measure the timing differences.
The AFU stays idle until an address is written into the predefined MMIO register. It then starts the clock counter and, depending on the mode, sends a read or write request to the address in main memory. The counter is stopped when a valid response to the request is received.

```
1  ./writeToUnSharedMemory [time] [attack] [write/read] [debug]
```

Listing 3.1: Specification of how to start the experiment

The timing measurements done by our clock counter can be read from a third MMIO register by the software implementation. We will discuss the first measurement results in section 3.1.3.

3.1.2 Software Implementation

Our software implementation is a command line tool written in C, used to control the AFU on the FPGA. We added command line parameters, that can be supplied when executing the program. The command to run the program is shown in listing 3.1. If no parameters are used, the default settings will be used with no output whatsoever.
The *time* parameter enables the timing measurement and printing the result to the command line. If omitted, access timings will not be measured. By using the *attack* parameter, the program prepares a preallocated memory page for shared access with the accelerator and then revokes the access rights from the accelerator again. Consequentially an unshared memory page will be accessed. This is already considered a DMA attack, as the FPGA accesses a memory page through DMA he should not be able to access. *Write* and *read* define what operation mode will be executed by the AFU; the default is read. And last but not least, the *debug* parameter ensures that additional info, helpful for debugging, is printed.
When the attack mode is not enabled, a proper data transfer will be initiated. That means a memory page with the systems default page size (in our case 4 KiB) will be allocated by the OPAE library and made non-swappable. Then the I/O virtual address for the shared buffer will be retrieved. Because there is no activated IOMMU, the referenced I/O address will be equivalent to the physical address of the buffer. This address is then written into the MMIO register of the FPGA. To measure the latency of the transaction on the software side, we use the time stamp counters *rdtsc* of the CPU, with one taken just before the address is written into the MMIO space and one taken after the content in memory changed. At last, the timing measurement from the FPGA is read.
There are a few changes with the attack mode enabled. We pre-allocate a memory page for ourselves, map it into the virtual address space of the accelerator and get the IOVA. Then we release the buffer from the FPGA to revoke its access rights on the memory. If we don't pre-allocate the memory page ourselves, but let the driver handle this, the memory page will be freed and not be accessible by software anymore. Because we still want to access it later from software, this can not happen. We shared the buffer with the FPGA

first, even though this is not necessary, so that we don't write to any memory location, which might lead to crashing the test platform.

After running the experiment, our expectations were confirmed by the results. We first analyzed the results of the non-attack mode and came to the following conclusions: (1) we can freely access our allocated memory page and (2) also access arbitrary other memory pages. This applies to the read and write mode. The explanation behind the first observation is obvious, while the second becomes clear when realizing that there is no mapping to domains or address translation, without an IOMMU. This means that no one performs access control on which physical addresses are being accessed by which device.
In the attack mode we observed that (3) we can still access the memory page that we unmapped from the FPGA's address space at will. This is as we expected, because this is just special case of the general observation (2).

To summarize the observations of our first experiments: apart from the IOMMU, there are no additional memory protection mechanisms that prevent us from reading and writing arbitrary memory locations from the FPGA. Furthermore, the FPGA can freely write and read any memory location.

To adapt our attack to closer resemble the threat model mentioned in section 1.2, we build another experiment with two software programs. One of these being the victim and the other being an attacker. These processes need to be run concurrently on the same computer. The victim allocates a memory page, prints out its physical address to the terminal and monitors the content of the page for a couple of seconds. The attacker program uses the physical address as an argument and writes it to the FPGAs MMIO like before.
The fact that our victim program gave us the exact physical address does not weaken the attack, because an attacker can start by reading through the whole memory until she finds the data that she is interested in. For our experiment we just wanted to make sure we were accessing a memory address we had control over.
As mentioned before, these attacks were working as expected and nothing new, but they gave us an understanding of the OPAE library, knowledge of how to develop and synthesize designs for FPGA and some timing measurements presented in the next subsection.

Table 3.1: Different measurements made by hardware and software on 1000 traces with disabled IOMMU. Software measured reads are omitted, because a read transaction is not detected by software.

measured by	mode	avg. cycles	min.	max.	standard deviation
Software	proper write	3354.07	2372	4692	399.60
Software	attack write	3289.66	2386	4672	402.97
Hardware	proper write	42.22	41	43	0.84
Hardware	attack write	42.22	41	43	0.86
Hardware	proper read	170.21	153	194	12.00
Hardware	attack read	170.77	154	195	11.93

3.1.3 Timing Measurements

To deduce whether measurements on software are relatable to measurements on hardware or if they are too susceptible to noise, timing measurements were made. Furthermore, they can add to our understanding of the performance impact of the IOMMU when comparing them to measurements taken with an enabled IOMMU.

In the results of all experiments, presented in this chapter, values that deviate from the average by more than 6 times the standard deviation are excluded from the analysis. These are outliers that were not reproducible and only distorted the measurements. This led to a loss of a maximum of 3 measured values, but usually none or only one was removed.

The measurements were taken on our test-server system with no other major processes running to minimize the amount of noise. There are no measurements made by software of read transactions. That's because the software does not realize when it's memory has been read by the FPGA. This is also the reason why DMA attacks are not detected by a victim's process if only data is extracted. Modifications of an applications' memory, on the other hand, are detectable, but that is quite hard to do since a regular integrity check would be needed to detect external changes in the programs memory.

The results of 1000 traces are presented in table 3.1. The average amount of cycles needed for the various transactions is different depending on the measuring device. There are multiple explanations for this. First, the software measurements are a lot higher than the hardware measurements, since the CPU is clocked at a higher frequency than the FPGA. The FPGA runs at 200 MHz, while the CPU is clocked at 2.2 GHz with 3.0 GHz turbo frequency. This causes the software measurement to be 11 to 15 times higher. From the point of view of the software the hardware measurements should average between 220 and 300 clock cycles long, which is not quite as fast as our measurements. There is quite a bit of fluctuation in the software measurements however, which is shown by the fact that the maximum recorded value is almost twice as high as the minimum. Whether this

can be derived from varying MMIO transaction times, other processes being scheduled on the CPU or varying latencies caused by the memory controller is unclear. Another possible cause for the timing issue might be interference from DDIO, which stores data from memory write transactions directly into the LLC. This results in faster access from the CPU compared to an access to memory. In general, it is difficult to detect, estimate and measure all causes of interference.

The hardware measurements of write transactions, on the other hand, deviate by such a small margin that this is basically negligible. This small deviation is due to the fact, that the response to the write transaction does not come from the memory controller, but from the FPGA Interface Unit when the signals have successfully been applied to the PCIe bus and the actual memory hasn't changed yet, as shown in fig. 3.1. This hypothesis is supported by [NAZ+18], as the PCIe memory write transaction is a posted transaction, which receives no acknowledgment of any kind from the memory controller. Since there is only one AFU on the FPGA and the interface is idle when no requests are send, the deviation between measurements is almost non-existent. Therefore these measurements are not significant and will not be considered in the future.

The read transactions recorded by hardware are fairly close together with a standard deviation of 12 cycles. However, while going through our data, we noticed that a result of 170 cycles was basically never occurring. Instead, we had a lot of measurements around 160 cycles and then again around 185, so we displayed all our data in histograms, seen in fig. 3.2. The measurements are split into two conglomerations. One reaches from 155 to 168 clock cycles and the other starts at 178 and goes until 194 clock cycles. The average value of 170 lies between these two peaks and does indeed occur very rarely in the measurements itself. At first, we thought that this separation takes place because data is sometimes stored in different levels of the CPU caches and sometimes in memory. After allocating a buffer, we write a 0 into the least-significant byte, before sharing the IOVA with the FPGA. This enables us to identify accurately when the DMA write transaction modified the contents of the RAM. Furthermore, it results in this memory block being cached in one of the CPU caches. The software then receives the results from cache when accessing the buffer. With DDIO enabled the memory write transaction will write directly into the Last-Level-Cache. This should result in the software noticing changes quicker than changes in memory. But as an explanation this doesn't make much sense, as we access this page only once and then free it. Thus, every execution of the test should get its results from a cached entry. Furthermore, we tested initializing the buffer only in write mode, as this is not necessary in read mode and increasing the sample size to 10,000, but the resulting histograms showed the same pattern. In the end we were not able to deduce why this phenomenon takes place.

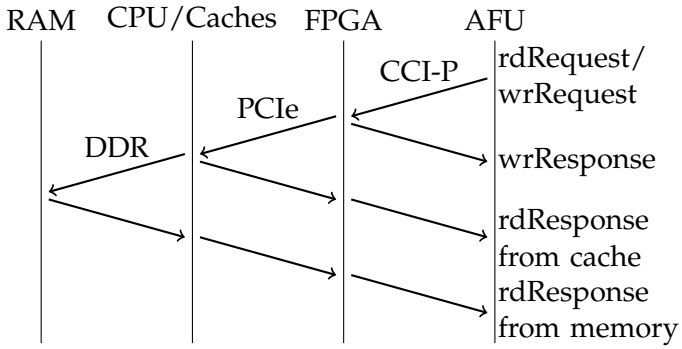

Figure 3.1: This figure shows the different components a transaction passes and where the response is returned that is seen by the FPGA.

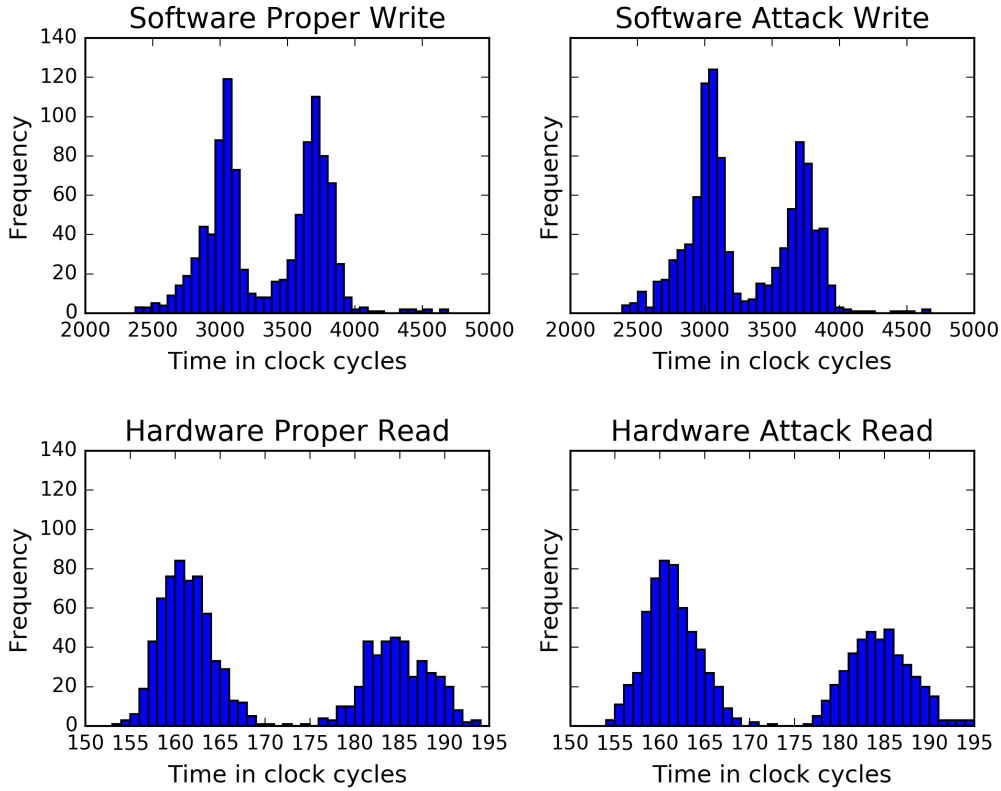

Figure 3.2: These histograms show the frequency of the measured access timings without IOMMU interference.

3.2 DMA Attack with IOMMU enabled

We used the same components, as in section 3.1, after enabling the IOMMU and executed our experiments again. Accessing memory outside our domain did not work anymore, as we expected. To do that, the software has to generate or guess an I/O virtual address and then share it with the FPGA. This address can not be translated by the translation structures mentioned in section 2.6.2, if it does not actually belong to a memory frame allocated to the device, resulting in a translation fault. Trying to access a physical address directly does also not work. Not only that, but our server automatically reboots, when we try to access a forged IOVA or PA. This behavior feels strange to us and the reason for why it is happening is unclear, but it is reproducible almost 100% of the time. A non-recoverable address translation fault typically requires the reset of the requesting device [Int19b, 127ff.], but, to our knowledge, should not result in a host system reboot, as this could be used as a Denial-of-Service (DoS) attack.

We tested for the spatial vulnerability explained in section 1.3. The driver allocates aligned memory pages that were a multiple of 4 KiB in size. This means that the whole mapped memory region is reserved for our process and no other process can store data in it. Allocating a memory page ourselves using *mmap* also resulted in aligned pages, where all entries had been initialized to zeroes, to not leak previous contents. We tried using other system calls to allocate smaller regions of memory, but the OPAE library declines pre-allocated memory pages that are not a non-zero multiple of the system's default page size. We therefore concluded that the spatial vulnerabilities have been fixed, most likely by enforcing alignment and certain sizes for memory pages in the OPAE library and the underlying FPGA driver.

Using the same program as before, i.e. mapping a memory page into the I/O virtual address space and then removing it again before accessing it for the first time, resulted in a non-successful attack and most of the time a reboot. However, by modifying our program to access the memory page once from the FPGA before unmapping it, we ensured that the translation data was cached in the IOTLB. We were then able to keep accessing the memory page, until the next IOTLB invalidation was done. This is due to the fact that the IOMMU does not maintain consistency between the IOTLB and the page tables. Instead, this responsibility was handed to the OS. The IOTLB entry did not get invalidated immediately, because invalidations of IOTLB entries are expensive. When the IOMMU runs in *deferred mode*, which is the default configuration for the Ubuntu distribution, the OS queues pending invalidations and whenever the queue is full or a timeout occurs this queue is processed. In *strict mode* the invalidation process is done immediately. We did not find any information about the length of this time frame between scheduled invalidations.

Table 3.2: Measurements across 1000 traces with enabled IOMMU and addresses not being cached in the IOTLB previous to access. Attack measurements are missing, because the attack does not work without cached addresses.

measured by	mode	avg. cycles	min.	max.	standard deviation
Software	proper write	4832.69	2810	8880	1292.15
Software	attack write	—	—	—	—
Hardware	proper read	310.84	212	581	108.13
Hardware	attack read	—	—	—	—

This experiment shows that our system still has a temporal vulnerability and this has not been fixed by operation system developers. A scenario where this might lead to data leakage is, when the OS quickly reuses a recently unmapped page for another process that places sensitive data in it while the FPGA still has access to that page through the translation cache mapping. However, this might be difficult to abuse by an attacker because she would need to know the IOVA to use the FPGA to access it. Probing for the address did not work in our experiments, since every faulty access has resulted in a system reboot.

3.2.1 Timing Measurements

Table 3.2 and fig. 3.3 show our measurements over 1000 traces with an enabled IOMMU and no caching of the address in the IOTLB previous to measuring the access timing. The attack did not work in this scenario, which is why these spots are crossed out in the table and not presented as histograms. The timings of write transactions measured on hardware are not expressive, as mentioned before, and are therefore excluded.
Looking at the software measurements of memory write transactions, one can see that these increased on average by almost 1500 cycles, compared to the previous values without IOMMU shown in table 3.1. Although an increase was expected due to the additional abstraction layer and the associated page walks, it is quite large at almost 50%. However, if we look at the distribution of the measurements in fig. 3.3, we see that the first peak in the measurements got shifted up by 800 cycles and the second peak moved from around 3750 to 4750. Both of them lost a bit in frequency though, with the addition of new smaller peaks around 6300 and 7250, resulting in a much wider interval than before. This is also reflected in the standard deviation, which is now 3-times larger than before. We think this might be due to the required page walks and the state of the translation caches, that were briefly mentioned in section 2.6.2. The IOTLB should not have influenced these measurements, as the memory page was never accessed over its IOVA previous to the measured access. It could only have been cached if the driver of the FPGA created an entry for that page when mapping it into the I/O virtual address space. In our eyes this is unlikely.

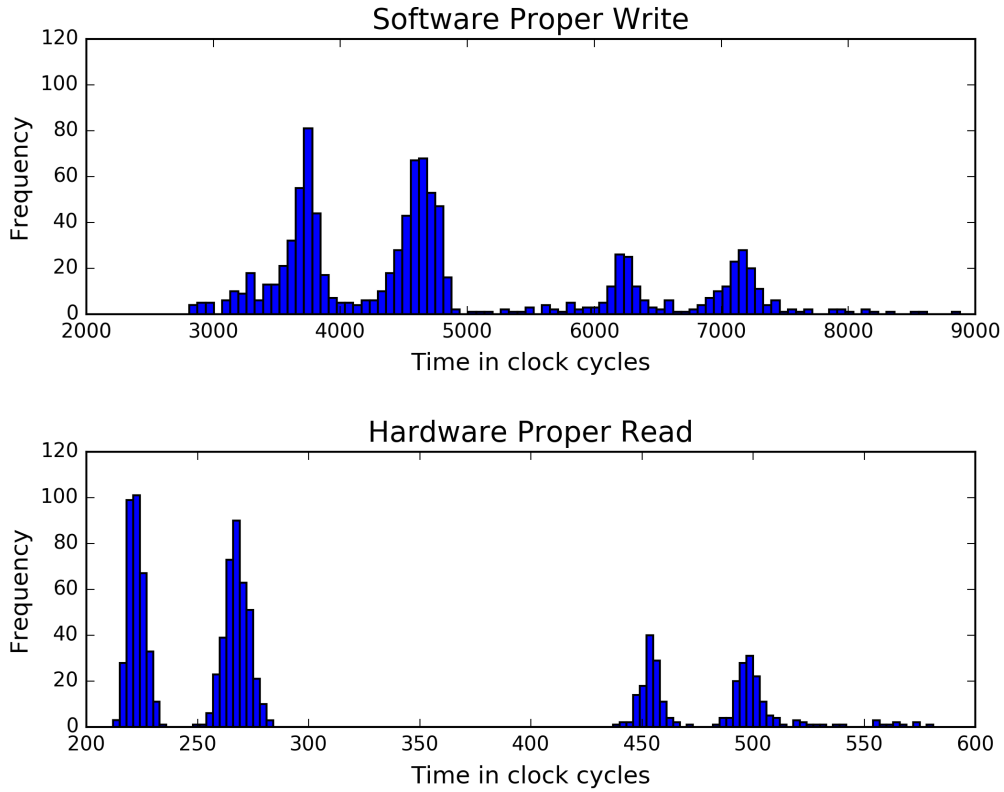

Figure 3.3: Histogram of the measurements done with IOMMU but no address caching.

When comparing the measurements from hardware to the ones from software in fig. 3.3, one can see a very similar pattern. Although the gap between peak 2 and 3 is wider in the hardware measurement, there are still four agglomerations and the expected increase in latency is clearly visible. The general increase seems to be around 65 clock cycles in hardware, as the first two peaks contain the most common measurements.

If we compare the values from before enabling the IOMMU table 3.1 and after enabling it and forcing address caching table 3.3, it is visible that the read measurements went back closely to their former average values we measured without IOMMU, if not even a little bit faster. This does not make sense because this would require for an IOTLB lookup and the subsequent memory access to be faster than a sole memory access. Nevertheless, this is also visible in fig. 3.4, but as the difference is so small, we assume this is an inaccuracy in our measurements.
One very important insight from comparing the histograms with and without address caching (fig. 3.3 & fig. 3.4) is the fact that accessing the IOVA once, before measuring the latency, decreased the access time of the first two peaks from 225 and 265 clock cycles to 158 and 180 respectively. This is the case, because the mapping of IOVA to PA has been cached in the IOTLB through our previous access. According to our experiments measured by the FPGA, this speeds up the translation process by 60 to 80 clock cycles. Thus, the added layer of abstraction through the IOMMU is not noticeable for 4 KiB pages, once the address mapping is cached when comparing it to the previous measurements shown in table 3.1. The two peaks furthest to the right from fig. 3.3 were completely removed by address caching. This supports our hypothesis that these were caused by the other translation caches of the IOMMU. In a scenario where the address mapping is already stored in the IOTLB, the state of these caches becomes obsolete.
However, there are still some inconsistencies we were not able to deduce. We did not find out why there are still two distinct peaks in our hardware measurements with address caching (fig. 3.4) and not only one. Although the software measurements suggest this pattern, it seems to be more specific to read accesses, as previously patterns were more distinguishable. One possible explanation might be that some of the read requests are answered from the LLC, while the other ones require a fetch from main memory, as shown in fig. 3.1. This pattern will not occur when the FPGA sends a write request as that data is always written to memory or the LLC, if DDIO is supported. This somewhat fits to the histograms of the software measured write transactions with address caching, however, this is contradicted by the histograms of the write transactions without IOMMU interference (fig. 3.2), where this pattern is also clearly visible.

Table 3.3: Measurements across 1000 traces with enabled IOMMU and addresses being cached in the IOTLB previous to access.

measured by	mode	avg. cycles	min.	max.	standard deviation
Software	proper write	948.85	584	1632	136.25
Software	attack write	981.37	512	1676	125.79
Hardware	proper read	166.15	147	191	12.02
Hardware	attack read	168.14	152	192	11.90

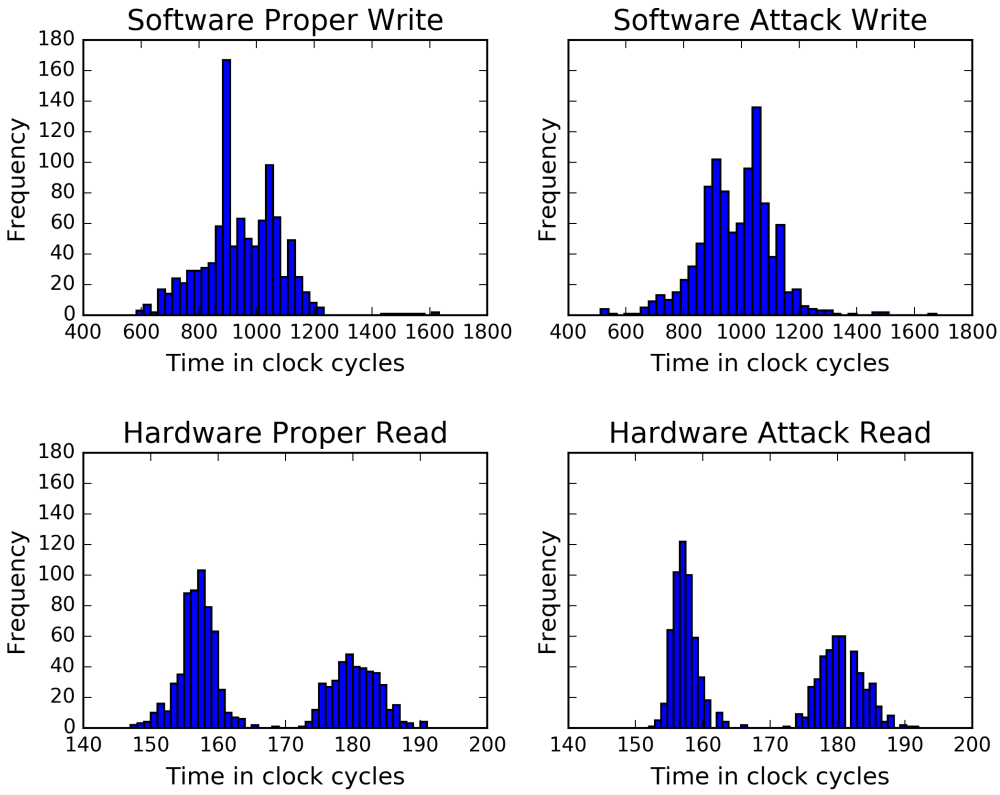

Figure 3.4: Histogram of the measurements done with IOMMU and forced address caching.

4 Reverse Engineering the IOTLB

The combination of a stricter driver, a stricter library and an IOMMU greatly complicates carrying out DMA attacks, as we have shown. However, the IOTLB brings a new side-channel that was not available before. The analyzed timing behavior might be exploitable to conduct a side-channel attack. Gras et al. [GRBG18] have shown that it is possible to carry out a side-channel attack through the TLB of the MMU. In their work, they reverse engineered the mapping function and the size of the Translation Look aside Buffer. Afterwards, they used their knowledge to build a timing-based side-channel attack on encryption schemes such as EdDSA and RSA. We want to show that this is also possible through the IOTLB.
The architecture of the TLB and the IOTLB is mostly kept secret by the hardware manufacturers, but for a successful side-channel attack, this knowledge is crucial. Thus, we started by running multiple tests to see when access timings would go up.

4.1 First Test

In out first test we wanted to see how many addresses are needed for access timings go up. To measure this we build a simple AFU, that, on receiving an IOVA by our software program, starts reading that memory line. As explained in section 3.1.3, only using the memory read transaction makes sense here, as we do not get an acknowledge from the memory controller for a completed memory write transaction. A software measured write transaction is too prone to fluctuations.
Our software for the first experiment can take 0 or 3 parameters. The program is run with the command shown in listing 4.1. If no parameters are supplied the default values 1, 128 and 1 are used. With this we get a default 4KiB memory page and an output formatted as *comma-separated values* (CSV) that can be exported into a file to analyze in a spreadsheet program. We took 128 pages as the default number of pages, because we have seen changes in timings at a slightly lower number and this gives us a bit more data to analyze which addresses are evicted.

```
./IOTLB_measurements [multiplier to page size, #pages, 1 = csv/0 = table output]
```

Listing 4.1: Specification of how to start the experiment

```
1 for (i = 0; i < num_of_pages; i++) {
2    iovaArray[i] = sharePage(accel_handle);
3
4    for (j = 0; j <= i; j++) {
5       clock_cntr = measureTime(accel_handle, iovaArray[j]);
6       printResult(iovaArray[j], clock_cntr);
7    }
8 }
```

Listing 4.2: Core logic of our first test program.

The core logic of our software program can be seen in listing 4.2. On startup, the program allocates the first memory page and maps it into the FPGA's virtual address space (line 2). Additionally, the IOVA is stored in an array, where all the addresses of this run will be stored. Then in line 5 the address will be written into the MMIO space of the FPGA. The FPGA then accesses the address and measures the timing. In line 6 the result will be printed to the command line. The program resumes by allocating another memory page (line 2) and then sending all previously stored addresses to the FPGA one by one, followed by the latest address (line 5). This goes on until the specified amount of pages is reached.

4.1.1 Results

The results are displayed in table 4.1. As one can see, the first time an address is added it takes around 260 to 275 cycles to read from that memory page. This is expected because the address is not cached in the IOTLB yet. This coincides with our results from section 3.2.1. Another important observation is that from address 119 onward every access to these addresses is in the range of 255 to 275 clock cycles, which overlaps with the uncached measurements. It should be noted that the access timings of address number 1 also increase from iteration 120 onward. Yet, this does not apply to address number 2 from run 121 forward, as one might expect if a *First-In-First-Out* (FIFO) or *Least-Recently-Used* (LRU) eviction strategy is used. Our experiment shows, that the first address is evicted for address 118 and all further addresses are not getting cached in the IOTLB. This led us to the conclusion that another eviction strategy is used and that the IOTLB can store 117 randomly taken IOVA addresses because this is reproducible over many runs.

Table 4.1: Excerpt from the measurements of the first two allocated addresses and the addresses at index 116-124. With every run a new address is added.

run \ address	1	2	...	116	117	118	119	120	121	122	123	124
117	176	179	...	263								
118	173	185	...	176	274							
119	175	180	...	179	184	269						
120	258	178	...	181	182	186	273					
121	261	178	...	181	184	182	265	261				
122	263	179	...	176	179	183	266	263	271			
123	259	177	...	176	182	187	272	257	270	262		
124	256	179	...	175	182	187	263	260	274	265	261	
125	263	176	...	178	181	184	268	257	271	260	259	266
126	257	178	...	177	180	185	272	259	266	259	255	256

After further inspection of the IOVAs sent to the FPGA, we realized that these are always the same. Starting from the same I/O virtual address and decreasing with every page by $0x1000$, i.e. 4096 in decimal notation, which is exactly the size of a 4 KiB page. In retrospect, this makes sense because the IOMMU abstracts the physical hardware layer by translating physical addresses into I/O virtual ones, creating the illusion of contiguous memory pages. This contiguous memory is represented by ongoing IOVAs. This means that the selected addresses were not as random as we first thought. It is also an explanation for why the result of 117 cached IOTLB entries is so reproducible. This number, however, is not very expressive as it does not necessarily relate to the wayness of the cache, since the memory pages were selected (in theory) randomly, but in practice not as much, and may not belong to the same cache set. That is why going forward, we tried to reverse engineer the mapping function, that maps the IOVA to the actual cache set.

4.2 Direct Mapping Hypothesis

Gras et al. [GRBG18] discovered that there are different mappings for L1 and L2 TLBs in Intel CPUs starting with the Skylake architecture. As we do not know if there are multiple levels in the IOTLB and which mapping is used there, we tested both their presented variants. The two general approaches are *direct* and *XOR mapping*. The first can be expressed as $cache\ set = page_{IOVA}\ mod\ s$, with s being the number of sets. XOR mapping uses a subset of $2p$ bits ($[12 + 2p{:}12]$) from the IOVA and consecutively XORs two of them together to receive a number consisting of p bits. This number is the cache set index of the address and maps to one of the 2^p sets. A more detailed explanation and our results are given in section 4.3.

Since we do not have access to performance counters regarding caches misses like the authors of [GRBG18] had, we have a higher barrier of entrance to reverse engineer the mapping function. Thus, we had to find a way to deduce the size of the IOTLB by only sending valid read requests from user space. We have tried this by building an *eviction set* for an arbitrary cache line for each of the reasonable combinations of sets s and ways w. An eviction set is a set of IOVAs that map to the same cache set and is large enough to evict every entry in that set. Vila et al. [VKM19] propose a test to determine if an eviction set works. Their approach is accessing all the elements of the eviction set once and measuring the time of each individual access. Then, they immediately do this again and compare the measurements. The first iteration ensures that all the elements are cached, while during the second iteration all the cache misses can be counted. If an eviction occurred during the first iteration, it is shown by the access delay in the second round. That indicates that the eviction set works, however, this does not mean that its size is minimal. The problem of finding a minimal eviction set is equivalent to learning the wayness of a cache [LYG+15]. By trying out all the sensible combinations of sets and ways one will find a minimal eviction set eventually.
We tested all combinations of cache sizes up to 256 sets and 128 ways in our experiment. We selected those boundaries because, according to Gras et al. [GRBG18], the L2 TLB of Broadwell Xeon CPUs has 256 sets, which is the highest measured number by them. The largest amount of ways, seen by the authors, has been 12 ways. We have seen the first evictions in our previous experiment with 118 addresses, which is why we want to have that spot in our measurements.
The core logic of our program can be seen in listing 4.3. This program is run with every possible combination of sets and ways up to the boundaries mentioned above. The IOVA of the allocated and mapped memory page in line 1 is the starting point for our eviction set. Every other element of our set needs to map to the same *set* calculated in line 2. In the while-loop, we keep allocating pages (line 5 & 6) and checking if the remainder, after dividing the new address by the number of sets, equals the number we want for our set (line 8). If it matches, we add that address to our set (line 9). This goes on until the size of our eviction set reaches the number of ways we want to test for. After that, we start measuring the access time of every address after each other (line 16) and print out the result (line 17). This is not just done twice as Vila et al. [VKM19] suggest, but done multiple times to reduce inaccuracies (line 14).

```
1  iovaArray[0] = sharePage(accel_handle, ptr); //returns an IOVA
2  set = iovaArray[0] % num_of_sets;
3
4  while (iovaArray[num_of_ways - 1] == 0) {
5     ptr = mmap(0, pagesize, PROTECTION, FLAGS_4K, -1, 0);
6     uint64_t tmp = sharePage(accel_handle, ptr);
7
8     if ((tmp % num_of_sets) == set) {
9        iovaArray[i] = tmp;
10       i++;
11    }
12 }
13
14 for (int j = 0; j < repetitions; j++) {
15    for (int i = 0; i < num_of_ways; i++) {
16       clock_cntr = measureTime(accel_handle, iovaArray[i]);
17       printResult(iovaArray[i], clock_cntr);
18    }
19 }
```

Listing 4.3: Core logic of our test program for direct mapping.

4.2.1 Results

If our hypothesis of direct mapping is true, we should be able to deduce the wayness w of the cache from these access timings. The smallest eviction set, that reliably results in the eviction of a cache line, shows us the number of ways because an eviction set, that evicts exactly one of its own addresses, has $w + 1$ elements. The amount of sets can be concluded by the number of sets anticipated when creating the working eviction set. In case there are multiple eviction sets with the same amount of elements, the smallest amount of anticipated sets is the correct one.
We ran our experiment with every address being accessed fifty times (line 14 in listing 4.3), instead of just twice and exported our data again into csv-files. Every file contains the measurements of one combination of sets and ways from the interval mentioned above. To display the data in a heat map, we wrote a python script that goes through all the data files and counts the amount of suspected evictions, i.e. measurements above 215 clock cycles (evictions in these experiments are around 250 clock cycles), per iteration. This resulted in fifty frequencies per file. We then picked the most common frequency per file and stored it in a 2-dimensional array that is displayed as a heat map in fig. 4.1.
The outcome does not look as clear as our results from section 4.1.1, where we have reliably seen 117 fast accesses. However, that threshold is still clearly visible in the heat map. There is quite a bit of noise in the upper half of the heat map. The explanation is most likely that the more address are selected, the higher is the probability of having at least some addresses mapping to the same set.

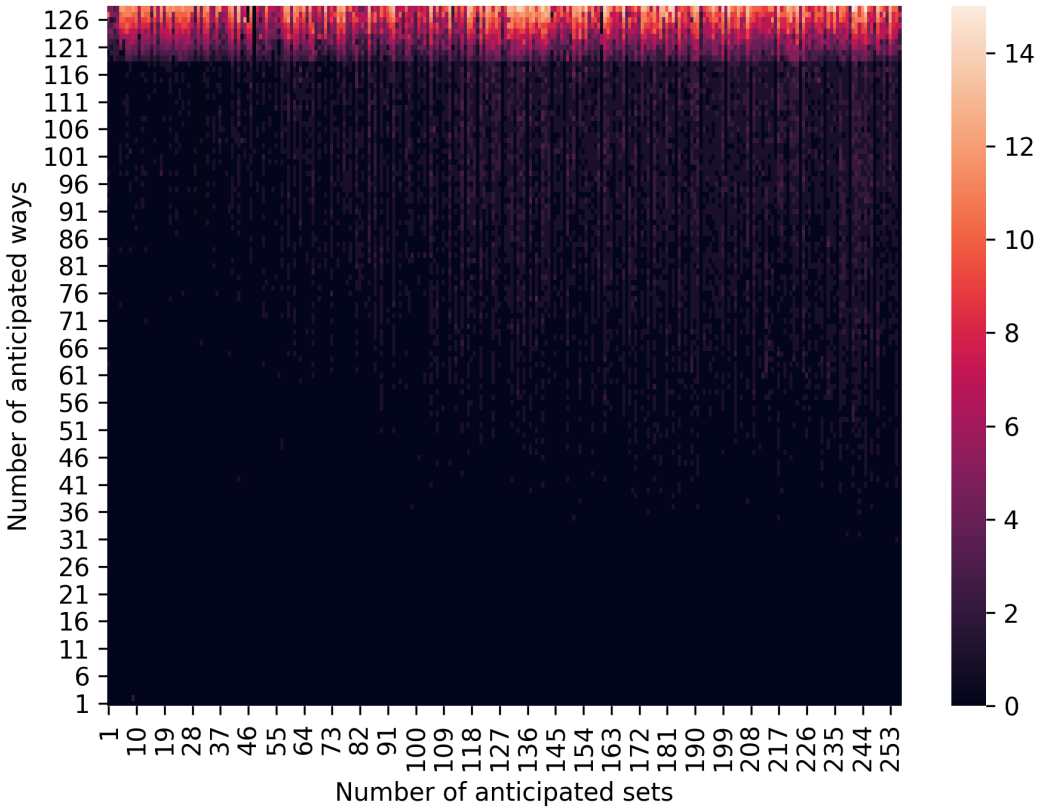

Figure 4.1: Heat map of the most common amount of evictions measured per combination with direct mapping.

Since the divider for the set index calculation is higher the more sets are anticipated, the addresses have more distance between them. This is the reason why the noise increase towards the right side in the heat map.

To compare the data to our previous experiment, we have to look at what has changed since then. First of all, our address selection follows the direct mapping hypothesis and is not just the first w addresses we allocated. And secondly, our access pattern changed from accessing address#1, then address#1 & address#2, then address#1, address#2 & address#3, and so on, to accessing address#1 to address#128 and then starting again. The different access patterns do not impact the measurements, as the same threshold exists as before. We have seen in previous experiments that one access to an address is enough to store the translation data in the IOTLB. The access pattern might change which addresses are evicted at cache contention, if the eviction policy takes the frequency of access into account. Nonetheless this should have no influence over the amount of fast accesses per iteration, as we only counted the general amount of evictions and not which addresses were evicted.

If our mapping hypothesis were correct, we could see multiple brighter colored lines running vertically down into the darker areas. This is due to the fact that we would have built a minimal eviction set for an arbitrary cache line that evicted one of its own entries. All the points in the same column of the graphic would also be eviction sets, but with more elements the further up the point is on the y-axis. Thus, the higher the point, the higher would be the number of evictions covered by this sent, resulting in a line running from dark to light. The values in the surrounding columns would be wrongly assumed set sizes. This means that we need on average more addresses to evict a cache line, as these addresses are spread across different cache sets. Therefore, these areas would still be darkly colored. This can only lead to one of two conclusions. Either our mapping hypothesis must be wrong or the IOTLB is a cache with around 117 ways. In our eyes this is very unlikely because a fully associative cache requires a lot more hardware than, for example, set-associative caches with less ways but more sets. This is due to the fact that multiple cache tags have to be checked at the same time to keep the latency of cache accesses low and thus multiple tag-compare-module are needed. The effort needed for implementing a 117 way seems too high to be actually considered by manufacturers.

This leaves us only with the theory that our prediction about the mapping function is wrong.

4.3 XOR Mapping Hypothesis

Going forward, we also tried the second mapping hypothesis mentioned by Gras et al. [GRBG18]. The *XOR-mapping* is best explained with an example: our cache has 16 sets ($2^p = 16 \rightarrow p = 4$) and maps 4KiB pages ($2^q = 4096 \rightarrow q = 12$). To map an address to one of these 16 sets we need the $2p$ bits after the page offset from the IOVA [$q + 2p - 1$:q], i.e. [19:12]. These bits will be XORed following the schematic below. The resulting 4 bit set_index determines the cache set.

$$IOVA[19] \oplus IOVA[15] = set\_index[3]$$
$$IOVA[18] \oplus IOVA[14] = set\_index[2]$$
$$IOVA[17] \oplus IOVA[13] = set\_index[1]$$
$$IOVA[16] \oplus IOVA[12] = set\_index[0]$$

With this hypothesis there is a much smaller space of possible combinations than with direct mapping. We can only address caches with set sizes that are a power of 2, because p has to be an integer. Thus we only test all powers of 2 up to 256.
The results can be seen in fig. 4.2. The pattern in this heat map is similar pattern to the previous one, where there is a clearly distinguishable increase in cache misses above 117 ways. Furthermore, the noise in the upper right hand corner is also there. The absence of bright vertical lines in the data indicates that this is also not the correct mapping function and the hardware manufacturers built something entirely different for the IOTLB.

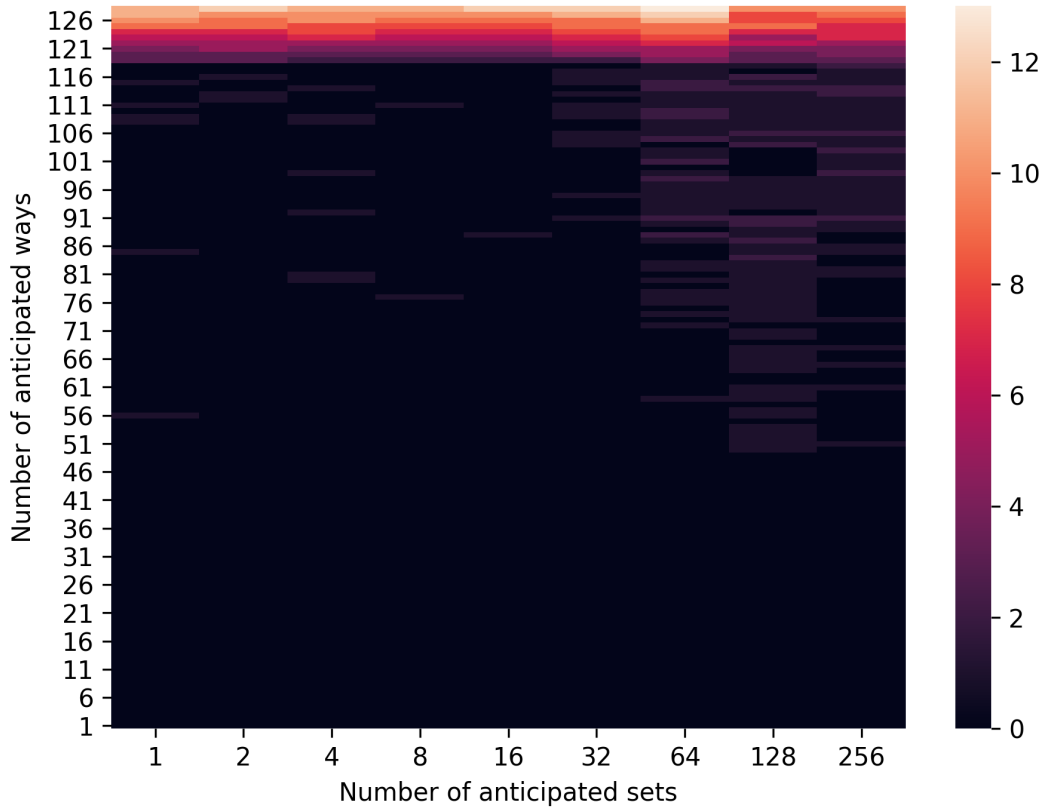

Figure 4.2: Heat map of the most common amount of evictions measured per combination with XOR mapping.

5 Conclusions

5.1 Summary

In chapter 3, we implemented a Direct Memory Access attack from an FPGA circumventing the IOMMU. We began measuring how long DMA read and write operations take on our system. These measurements were made by two different platforms and with an enabled and disabled IOMMU. The software's readings show how long it takes from sending the address to the FPGA to it changing the memory contents. On the hardware it measures how long it takes from the arrival of the address to the receipt of the confirmation. Our conclusion is that read transactions are best recorded from the FPGA and write transactions from the software. Furthermore, by comparing the measurements from before and after enabling the IOMMU, we have shown that it impacts the latency of read a transaction by ≈65 clock cycles, measured on the FPGA, and the latency of write transaction by ≈800 cycles, on the CPU. Additionally, we have proven that after accessing a memory page once, the translation data is cached in the IOTLB. Even after unmapping, this memory page can still be accessed by the FPGA, resulting in a successful DMA attack. However, there is no significant difference in the timing measurements and their distribution between a regular DMA transaction and our DMA attack.
Referring back to section 1.1, it has been shown that it is indeed possible to implement a DMA attack on current hybrid platforms. Although this is only possible, because operating system developers consciously decided to keep this vulnerability for performance reasons.
In chapter 4, we analyzed how many memory pages we need to allocate to force evictions from the IOTLB. The measurements from chapter 3 were used to decide which accesses have been evictions. The threshold lies around 117 addresses where we can reliably see the amount of evictions increase. Using the direct and XOR mapping hypotheses, we build an eviction set for each combination of sets and ways and measured their individual access timings. The results show that we can not deduce the wayness or the amount of sets of our cache. Thus, we failed in reverse engineering the IOTLB. Consequently, we were not able to verify the other two theses mentioned in section 1.1.

5.2 Discussion

Operating systems trusted peripheral unconditionally for a very long time. This has changed since the introduction of IOMMUs as memory protection mechanisms, but this trust is deeply rooted in many operating systems and drivers. This makes it hard for developers to change existing code to distrust peripherals. Even though they are mostly disabled by default, IOMMUs are definitely a step in the right direction.

The DMA attack shown by us can leak at most as much memory as the page was previously large. Increasing the page size to huge pages or even a 1 GiB page would most likely increase the practical relevance of the attack. However, this needs to be evaluated.

Many of the problems we had in the beginning of this thesis were caused by the FPGA Interface Unit. It made designing hardware much simpler, but also severely limited our options for implementing a DMA attack. This made us discard many ideas we had from the start. A less strict threat model would have definitely allowed more room to explore attacks through the PCIe protocol. Forging the source-id and using PCIe ATS are only two options we considered when starting working on this thesis.

We have presented some of the published DMA attacks of the recent years and are happy to see and read that many of them have been fixed. Thus we recommend anyone, using a computer with external DMA-capable interfaces or internal programmable devices, to enable their IOMMU. Furthermore, disabling PCIe Address Translation Services is also advised, as this completely undermines the IOMMU protection.

5.3 Future Work

In the future, one would have to test how much practical relevance our shown DMA attack has. We had a few runs where accessing the memory page was still possible for multiple seconds after unmapping and then again runs where our server automatically rebooted. The highest we tested were 20 seconds, but this should by no means be considered the upper boundary, as we did not run the experiment many times due to the rebooting issue. This issue should be examined in particular to make further experiments with DMA attacks less time and nerve consuming.
Using an FPGA without an interface unit that abstracts the PCIe layer from the hardware design is another thing to be considered. This enables an attacker to directly build PCIe packets and thus find design flaws or implementation errors in the PCIe protocol. One idea we had from the beginning was forging the source-id used in the first level address

translation. However, we were not able to implement this, because we couldn't modify PCIe headers due to the Intel FIU. The reason for not removing the FIU were limitations set by our threat model.

As we were not able to successfully reverse engineer the size of our cache, we propose expanding the search for the mapping function. A rather simple approach would be to continue increasing the number of sets tested with the XOR mapping. IOVAs have 48 bits, minus the 12 bits for the 4KiB page offset, there still remain 36 bits. This results in $2^{18} = 262144$ possibly addressable sets. It is rather unlikely that this is the number of sets in our cache, as this would mean the IOTLB is a directly mapped cache, but increasing the sets to 2048 or even further should be a good idea. Especially because this does not increase the runtime of the experiment by much. We were not able to conduct this experiment before the deadline. For the direct mapping, however, increasing the sets is much more time-consuming, as going from anticipating 256 sets to 512 doubles the number of performed tests. To speed this up, one might think about ignoring all the set sizes that are not a power of 2.
Changing the order in which bits of the IOVA are XOR-ed is another approach we recommend, but there are many possible combinations. A possible solution to that would be using huge pages. These use the 21-least significant bits of the I/O virtual address for the page offset. This results in 9 bits less to take into account when guessing the mapping function. Our system IOMMU also supports 1 GiB large pages, which use the 30-least significant bits, resulting in 18 IOVA address bits to XOR. Nonetheless, the space of possibilities is still very large.
When one is able to determine the size of the cache, another interesting thing to investigate is the replacement policy. Knowing the replacement policy gives an attacker the ability to predict which cache line is being evicted next. This knowledge can help using the IOTLB as a side-channel when attacking network interface cards (NICs), graphical processing units (GPUs) or other peripheral devices. For that to work, one would need to check whether cache contention between devices occurs. If that is the case, monitoring the translation cache accesses of a NIC might result in leaking website access patterns or typing patterns during an SSH-connection. Likewise, attacking a GPU might also be possible and give an attacker the possibility to predict screen content.
Another section of work is implementing the a covert channel for mutually cooperating parties. These can communicate by accessing IOTLB sets, which results in higher latency in the same set for the other party.
Basically any cache attack that works with page size-grained information can work.

References

[AAA19] Altaf Ahmed, Abdullah Aljumah, and Gulam Ahmad. Design and Implementation of a Direct Memory Access Controller for Embedded Applications. *International Journal of Technology*, 10:309, 04 2019.

[AD10] Damien Aumaitre and Christophe Devine. Subverting Windows 7 x64 Kernel with DMA attacks. `http://conference.hackinthebox.nl/hitbsecconf2010ams/materials/D2T2%20-%20Devine%20&%20Aumaitre%20-%20Subverting%20Windows%207%20x64%20Kernel%20with%20DMA%20Attacks.pdf`, July 2010. Hack In The Box Security Conference 2010 - Amsterdam (accessed 30.07.2020).

[Adv16] Advanced Micro Devices. *AMD I/O Virtualization Technology (IOMMU) Specification*, 3.00 edition, December 2016.

[Ama] Inc. Amazon.com. Amazon EC2 F1-Instances. `https://aws.amazon.com/de/ec2/instance-types/f1/`. (accessed 26.07.2020).

[Anu16] Murali Anumothu. Design and Analysis of DMA Controller for System on Chip based Applications. *International Journal of VLSI and Embedded Systems-IJVES*, 07:1685–1690, 06 2016.

[ARM16] ARM Limited. *ARM System Memory Management Unit Architecture Specification*, 2.0 edition, June 2016.

[BDK05] Michael Becher, Maximillian Dornseif, and Christian Klein. Firewire: all your memory are belong to us. In *Proceedings of CanSecWest*, 01 2005.

[Boi06] Adam Boileau. Hit by a Bus: Physical Access Attacks with Firewire. In *Ruxcon 2006*, May 2006.

[BS12] Rory Breuk and Albert Spruyt. Integrating DMA attacks in exploitation frameworks. Technical report, University of Amsterdam, 2012.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016. `https://eprint.iacr.org/2016/086.pdf` (accessed 30.07.2020).

[Cor] Intel Corporation. OPAE C Library Documentation. https://opae.github.io/0.13.0/docs/install_guide/installation_guide.html (accessed 21.07.2020).

[CS15] Vibhu Chinmay and Shubham Sachdeva. Review Paper: Design of basic DMA Controller Using VHDL. 2015.

[Dor] Maximillian Dornseif. Owned by an iPod, November. PacSec 2004 Japan. https://web.archive.org/web/20071011191205/http://md.hudora.de/presentations/firewire/PacSec2004.pdf (accessed 30.07.2020).

[ECS] Alibaba Cloud ECS. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057. (accessed 26.07.2020).

[Fal14] Heiko Falk. Lecture slides in Grundlagen der Rechnerarchitektur - Ein-/Ausgabe. Institut für Eingebettete Systeme/Echtzeitsysteme of the University Ulm, October 2014.

[FC] Joe FitzPatrick and Miles Crabill. Stupid PCIe Tricks - featuring NSA Playset: PCIe. https://milescrabill.com/files/playset-pcie.pdf. (accessed 19.07.2020).

[Fit] Joe FitzPatrick. SLOTSCREAMER. https://github.com/NSAPlayset/SLOTSCREAMER (accessed 10.12.2019).

[For11] Jeff Forristal. Hardware Involved Software Attacks. https://forristal.com/material/Forristal_Hardware_Involved_Software_Attacks.pdf, December 2011. CanSecWest Vancouver 2012.

[Fri] Ulf Frisk. PCILeech. https://github.com/ufrisk/pcileech (accessed 10.12.2019).

[GBAS] Scott Gibbons, Jonathan Buie, Michael Adler, and Gabriel Southern. Intel FPGA Basic Building Blocks (BBB). https://github.com/OPAE/intel-fpga-bbb/tree/master/samples/tutorial/01_hello_world. (accessed 11.06.2020).

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB At-

tacks. In *27th USENIX Security Symposium*, August 2018. Pwnie Award Nomination for Most Innovative Research.

[Har94] Audrey F. Harvey. DMA Fundamentals on Various PC Platforms. 1994.

[Int15] Intel Corporation. *82C37A CMOS High Performance Programmable DMA Controller Datasheet*, 4.0 edition, October 2015.

[Int17] Intel Corporation. *Intel® 200 (including X299) and Intel® Z370 Series Chipset Families Platform Controller Hub (PCH) Datasheet - Volume 1 of 2*, 003 edition, October 2017.

[Int18] Intel Corporation. *Intel® 200 (including X299) and Intel® Z370 Series Chipset Families Platform Controller Hub (PCH) Datasheet - Volume 2 of 2*, 006 edition, February 2018.

[Int19a] Intel Corporation. *Intel Acceleration Stack for Intel® Xeon® CPU with FPGAs Core Cache Interface (CCI-P) Reference Manual*, November 2019. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf` (accessed 21.07.2020).

[Int19b] Intel Corporation. *Intel® Virtualization Technology for Directed I/O - Architecture Specification*, 3.1 edition, June 2019.

[Int20] Intel Corporation. *Intel ® Trusted Execution Technology (Intel ® TXT)*, 016.1 edition, May 2020.

[KGA+20] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *41st IEEE Symposium on Security and Privacy (S&P'19)*, May 2020. Intel Bounty Reward.

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium*, pages 189–204, Bellevue, WA, 2012. USENIX.

[Kup18] Gil Kupfer. IOMMU-resistant DMA attacks. master thesis, Israel Institute of Technology, 2018.

[LGY+16] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel

attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, 2016.

[lin] Linux kernel documentation for intel iommu support. `https://www.kernel.org/doc/Documentation/Intel-IOMMU.txt`. (accessed: 18.07.2020).

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *36th IEEE Symposium on Security and Privacy (S&P'15)*, pages 605–622, 2015.

[MEANK16] Benoît Morgan, Éric Alata, Vincent Nicomette, and Mohamed Kaâniche. Bypassing IOMMU Protection against I/O Attacks. In *7th Latin-American Symposium on Dependable Computing (LADC)*, pages 145–150, 2016.

[MEANK18] Benoît Morgan, Éric Alata, Vincent Nicomette, and Mohamed Kaâniche. IOMMU protection against I/O attacks: a vulnerability and a proof of concept. In *Journal of the Brazilian Computer Society*, volume 24, 2018.

[MRG+19] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, February 2019.

[NAZ+18] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES". In *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[PCI09] PCI-SIG ®. *Address Translation Services*, 1.1 edition, January 2009.

[PCI10] PCI-SIG ®. *PCI Express® Base Specification*, 3.0 edition, November 2010.

[SB13] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 21–41, Berlin, Heidelberg, 2013. Springer-Verlag.

[SELND10] Fernand Lone Sang, Éric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. In *5th International Conference on Malicious and Unwanted Software*, pages 7–14, 2010.

[SND11] Fernand Lone Sang, Vincent Nicomette, and Yves Deswarte. I/O Attacks in Intel PC-based Architectures and Countermeasures. In *2011 First SysSec Workshop*, pages 19–26, July 2011.

[sr14] snare and rzn. Thunderbolts and Lightning - very, very frightening. `https://www.youtube.com/watch?v=0FoVmBOdbhg`, April 2014.

[Tur14] Jim Turley. Introduction to Intel® Architecture. Technical report, Intel Corporation, 2014.

[VKM19] Pepe Vila, Boris Köpf, and José F. Morales. Theory and Practice of Finding Eviction Sets. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, pages 39–54, 2019.

[WRT09] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Another Way to Circumvent Intel ® Trusted Execution Technology. December 2009.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[YZZ17] Jiewen Yao, Vincent J. Zimmer, and Star Zeng. A Tour Beyond BIOS: Using IOMMU for DMA Protection in UEFI Firmware. Technical report, Intel Corporation, 2017.